How To Code V7 - Edited by Comrade J / SAE

AmigaGuide Release 7/July/93

About HowToCode

********** About HowToCode **********

How to write demos that work (Version 7) - 7/Jul/93

(or the Amiga Demo Coders Reference Manual)

Edited by Comrade J/SAE

Co-Editor Carl-Henrik Skårstedt (Asterix/Movement)

email:comradej@althera.demon.co.uk

Merged to single document for PDF by XXXXX

NEW AMIGAGUIDE FORMAT -

Due to numerous requests HowToCode has now been rewritten into
AmigaGuide format! It's still readable as text files (just!)
if you haven't got AmigaGuide, but download it and use it, it's good!

* Please note this is a REPLACEMENT to text files howtocode1.txt
through howtocode6.txt. Sysops, please remove these earlier files
as they contain many mistakes. Thanks in advance...*

Thanks to:

Richard Aplin, Matthew Arnold, Dean Ashton, Andreas Axelsson
Michael Bauer, Bilbo the First/Hypenosis, Eddy Carroll, Mark Cassidy,
Nils Liaaen Corneliusen, Walter Dao, David Dustin, Chris Green,
Joep Grooten, Grue, Jim Hawkins, Arno Hollosi, Lars Holmgren,
Philip Jespersen, Irmen de Jong, Marius Kintel, Christopher Klaus,
Mark G Knibbs, Jesper R Larsen, Jacco de Leeuw, Tero Lehtonen,
Patrik Lundquist, Jonas Matton, Stanley Merkx, Jesse Michael,
Jonas Minnbergh, Richard Moeskops, John Derek Muir, Marco Nelissen,
Karsten Niemeier, Boerge Noest, David Nouls, Mats Olsson, Dave Parkinson,
Andrew Patterson, Raymond Penners, Vidar Petursson, Michael Pollard,
Jolyon Ralph, Bjorn Reese, Vic Ricker, Timo Rossi, Michael Ryffel,
Conrade Sanderson, Doz/Shining, Toby Simpson, Darrell Tam, Lehtonen Tero,
Magnus Timmerby, Yee Tom, and Michel Vissers.

for their comments and contributions, both rude and polite!

And Commodore - Pull your fingers out and **DISTRIBUTE** the V39
docs to *all* Amiga programmers.

Please note that this file is the work of many authors, although
the spelling mistakes are all my fault!

I apologise in advance for mistakes in HowToCode, I am unable to check
everything thoroughly and certainly not on all machines, and in
particular with AGA information so much misleading and contradictory
information is going around that it is inevitable that mistakes will
happen. Please point these out to me, however trivial! I promise
I won't be angry if 100 people point out the same mistake. If no
one does and everyone tells me later that they knew - then I will!

distributed in any electronic form. The copyright of contributions quoted from other authors remains with the original author. If you would like to contribute to this file, email me at the address below...

If you intend to use part or all of HTC in a Public Domain disk magazine (or similar), please email me first. I *will* give permission freely, but I want to make sure that only the latest versions are published. Please also mention your request to Carl-Henrik, especially if it concerns Vectors.txt since it will be continually updated and debugged for some time from now.

The startup code in this article is freeware and may be used by anyone for any purpose.

All trademarks and registered names (Workbench, Kickstart, etc) acknowledged.

All opinions expressed in this article are my own, and in no way reflect those of anyone else. Please note that many of the programming practices described in this text are ONLY applicable for demo coding, and should not be used for Games and other programming.

I didn't write this for fun, I wrote it for you to use!

If you want to make a contribution please email it to me:
I prefer plain ASCII set to no more than 75 column width, and no tabs if possible (although I can fix text sent to me..)

If you strongly disagree with anything I write, or you want to send me some source or demos to test on Amiga 1200/4000 etc, or you have questions about Amiga programming, or suggestions for future articles, or just want to chat about the best way to optimise automatic copperlist generation code, then contact me via email at:

comradej@althera.demon.co.uk.

********** 0. Beginners ***********

How do I begin?

If you are just starting to learn programming, and you want
a good place to begin learning assembler, buy Amos!. It's
very easy to write assembler code, load it into amos and test it, and
if you end up being hopeless with assembler, you can write your
demos in AMOS instead! :-)

For example, take this routine:

```
;simplemaths.s

        add.l  d0,d1    ; add contents of d0 to d1
        rts
```

Assemble this with Devpac and what do you get? Not a lot.

Now, load AMOS and type this:

```
Pload "ram:simplemaths",1  ' load executable file into bank 1
Input "Enter a number ";n1
Input "Enter another number ";n2
dreg(0) = n1               ' Store n1 in 68000 register d0
dreg(1) = n2               ' Store n2 in 68000 register d1
call(1)                    ' Run your machinecode routine
Print n1;" plus ";n2;" equals ";dreg(1)    ' returns result in d1
```

You can start playing with 68000 instructions this way, seeing how
they work, without having to 'jump in the deep end' writing
routines to set up displays, copperlists, windows or writing to
the console.

You can also pass your machine code the address of AMOS's
bitplanes (by Phybase(0) to Phybase(n) where n is number of
bitplanes - 1), so you can write your own vector/bob code
and test it easily before writing your own front end code.

Once you have got the hang of 68000, you can drop Amos.

Another good way is to write some code in C, and use the inline
debugging options with SAS C, and OMD to examine what your C
compiler actually generates. To do this with SAS V6.x do the
following

```
SC debug=full myprog.c

OMD >ram:omdoutput myprog.o myprog.c
```

You will get each line of C code interleaved with the assembler
that it generates. Very handy!

It's also amazing how good the code generated by SAS C 6.2 really
is.

********** 1. Introduction **********

This file has grown somewhat from the file uploaded over Christmas 1992. I've been very busy over the last two months, so I'm sorry that I haven't been able to update this sooner. It started as an angry protest after several new demos I downloaded refused to work on my 3000, and has ended up as a sort of general how-to-code type article, with particular emphasis on the Amiga 1200.

Now, as many of you may know, Commodore have not released hardware information on the AGA chipset, indeed they have said they will not (as the registers will change in the future). Demo coders may not be too concerned about what is coming in a year or two, but IF YOU ARE WRITING COMMERCIAL SOFTWARE you must be.

Chris Green, from Commodore US, asked me to mention the following:

"I'd like it if you acknowledged early in your text that it IS possible to do quite exciting demos without poking any hardware registers, and that this can be as interesting as direct hardware access. amiga.physik.unizh.ch has two AGA demos with source code by me, AABoing and TMapdemo. These probably seem pretty lame by normal demo standards as I didn't have time to do any nifty artwork or sound, and each only does one thing. but they do show the POTENTIAL for OS friendly demos."

I have seen these demos and they are very neat. Currently you cannot do serious copper tricks with the OS (or can you Chris? I'd love to see some examples if you can...), for example smooth gradiated background copperlists or all that fun messing with bitplane pointers and modulos. But for a lot of things the Kickstart 3.0 graphics.library is very capable. If you are in desperate need for some hardware trick that the OS can't handle, let Chris know about it, you never know what might make it into the next OS version!

Chris mentions QBlit and QBSBlit , interrupt driven blitter access. These are things that will make games in particular far easier to write under the OS now.

Chris also says "Note that if I did a 256 color lores screen using this document, it would run fifty times slower than one created using the OS, as you haven't figured out enhanced fetch modes yet. A Hires 256 color screen wouldn't even work."

There are some new additions to the AGA chapter that discuss some of this problem, but if you want maximum performance from an AGA system, use the OS.

Remember that on the A1200 chipram has wait-states, while the 32-bit ROM doesn't. So use the ROM routines, some of them run faster than anything you could possibly write (on a A1200 with just 2Mb ram).

The only drawback is again documentation. To learn how to code V39 OS programs you need the V39 includes and autodocs, which I'm not allowed to include here.

Perhaps, in a later release, I'll give some highlights of V39 programming... Get Chris Green's example code, it's a good place to start.

Register as a developer with your local Commodore office to get the autodocs and includes, it's relatively inexpensive (£85 per year in the UK). You

can now buy the includes/autodocs and Rom Kernal manuals in AmigaGuide from Commodore US on a CD-ROM (developers only), the CATS CD Edition 2. It's excellent!

Most demos I've seen use similar startup code to that I was using back in 1988. Hey guys, wake up! The Amiga has changed quite a bit since then.

Thanks to everyone who has replied. Any more questions, queries, or "CJ/CH, you got it wrong again!" send your email to the following address:

comradej@althera.demon.co.uk

********** 2. General Guidelines **********

General Guidelines for Amiga Demo Programming

1. Read the Manuals
2. Self-Modifying code
3. Use Relocatable code
4. All addresses are 32bit
5. Packers/Crunchers
6. Avoid unnecessary hardware access
7. Opening libraries properly
8. Nothing is fixed - almost!
9. Version Numbers
10. System private structures/functions

********** 2.1 General Guidelines - Read the Manuals ***********

Read the Commodore manuals. All of them. Borrow them off friends or from
your local public library if you have to.

Read the "General Amiga Development Guidelines" in the dark grey (2.04)
Hardware Reference Manual and follow them TO THE LETTER.
If it says "Leave this bit cleared" then don't set it!

The official manuals are currently (up to 2.04 revision)

Addison Wesley

Amiga Rom Kernal Manual - Libraries
Amiga Rom Kernal Manual - Devices
Amiga Rom Kernal Manual - Includes and Autodocs
Amiga Hardware Reference Manual

(All the above are available on CD from CATS USA)

Amiga Style Guide

Bantam

AmigaDOS Manual 3rd Edition

A V39/V40 Kickstart update manual should be available shortly.

********** 2.2 General Guidelines - Self Modifying Code **********

Don't use self-modifying code. Processors with cache ram cannot handle self-modifying code at all well. They grab a large number of instructions from ram in one go, and execute them from cache ram. If these instructions alter themselves the changes are not made to the copy in cache ram, so the code can crash. The larger the cache the more likely this is to happen, even when you think you will be safe, so the best strategy is to either

a) Disable caches (and suffer a large speed-loss penalty)

b) Avoid using self modifying code.

********** 2.3 General Guidelines - Write Relocatable Code **********

If you write demos that run from a fixed address you should be shot.
NEVER EVER DO THIS. It's stupid and completely unnecessary.
Now with so many versions of the OS, different processors, memory
configurations and third party peripherals it's impossible to
say any particular area of ram will be free to just take and
use.

Remember that writing relocatable code does *NOT* mean you
have to write PC relative code. Your assembler and AmigaDOS
handle all the relocation for you at load time.

It's not as though allocating ram legally is dificult. If you
can't handle it then perhaps you should give up coding and
take up graphics or something :-)

If you require bitplanes to be on a 64Kb boundary then try the
following (in pseudo-code because I'm still too lazy to write it
in asm for you):

```
        for c=65536 to (top of chip ram) step 65536

        if AllocAbs(c,NUMBER_OF_BYTES_YOU_WANT) == TRUE then goto ok:

        next c:

        print "sorry. No free ram. Close down something and retry demo!"
        stop

ok:        Run_Outrageous_demo with mem at c
```

Keep your code in multiple sections. Several small sections are
better than one large section, they will more easily fit in and run
on a system with fragmented memory. Lots of calls across sections
are slower than with a single section, so keep all your relevent
code together. Keep code in a public memory section:

```
        section mycode,code
```

Keep graphics, copperlists and similar in a chip ram section:

```
        section mydata,data_c
```

Never use code_f,data_f or bss_f as these will fail on a chipram
only machine.

And one final thing, I think many demo coders have realised this
now, but $C00000 memory does not exist on any production machines
now, so stop using it!!!

********** 2.4 General Guidelines - 32Bit Addresses ***********

Always treat *ALL* addresses as 32-bit values.

"Oh look" says clever programmer. "If I access $dcdff180 I can access the colour0 hardware register, but it confuses people hacking my code!".

Oh no you can't. On a machine with a 32-bit address bus (any accelerated Amiga) this doesn't work. And all us hackers know this trick now anyway :-)

Always pad out 24-bit addresses (eg $123456) with ZEROs in the high byte ($00123456). Do not use the upper byte for data, for storing your IQ, for scrolly messages or for anything else.

Similarly, on non ECS machines the bottom 512k of memory was paged four times on the address bus, eg:

```
        move.l #$12345678,$0

        move.l        $80000,d0        ; d0 = $12345678
        move.l        $100000,d1        ; d1 = $12345678
        move.l        $180000,d2        ; d2 = $12345678
```

This does not work on ECS and upwards!!!! You will get meaningless results if you try this, so PLEASE do not do it!

********** 2.5 General Guidelines - Using Packers/Crunchers ***********

Don't ever use Tetrapack or Bytekiller based packers. They are crap.
Many more demos fall over due to being packed with crap packers than
anything else. If you are spreading your demo by electronic means
(which most people do now, the days of the SAE Demodisks are long
gone!) then assemble your code, and use LHARC to archive it, you
will get better compression with LHARC than with most runtime
packers.

If you *have* to pack your demos, then use Powerpacker 4+, Turbo
Imploder or Titanics Cruncher, which I've had no problems with myself.

(found in the documentation to IMPLODER 4.0)

>** 68040 Cache Coherency **
>
>With the advent of the 68040 processor, programs that diddle with code which is
>subsequently executed will be prone to some problems. I don't mean the usual
>self-modifying code causing the code cached in the data cache to no longer
>be as the algorithm expects. This is something the Imploder never had a
>problem with, indeed the Imploder has always worked fine with anything
>upto and including an 68030.
>
>The reason the 68040 is different is that it has a "copyback" mode. In this
>mode (which WILL be used by people because it increases speed dramatically)
>writes get cached and aren't guaranteed to be written out to main memory
>immediately. Thus 4 subsequent byte writes will require only one longword
>main memory write access. Now you might have heard that the 68040 does
>bus-snooping. The odd thing is that it doesn't snoop the internal cache
>buses!
>
>Thus if you stuff some code into memory and try to execute it, chances are
>some of it will still be in the data cache. The code cache won't know about
>this and won't be notified when it caches from main memory those locations
>which do not yet contain code still to be written out from the data caches.
>This problem is amplified by the absolutely huge size of the caches.
>
>So programs that move code, like the explosion algorithms, need to do a
>cache flush after being done. As of version 4.0, the appended decompression
>algorithms as well as the explode.library flush the cache, but only onder OS
>2.0. The reason for this is that only OS 2.0 has calls for cache-flushing.
>
>This is yet another reason not to distribute imploded programs; they might
>just cross the path of a proud '40 owner still running under 1.3.

I doubt it! Only a complete *IDIOT* would run an '040 under KS1.3. They
deserve to have their software crash!!

>It will be interesting to see how many other applications will run into
>trouble once the '40 comes into common use among Amiga owners. The problem
>explained above is something that could not have been easily anticipated
>by developers. It is known that the startup code shipped with certain
>compilers does copy bits of code, so it might very well be a large problem.

You can use the following exec.library functions to solve the problem.

   CacheClearU() and CacheControl()

Both functions are available with Kickstart 2.0 and above.

I strongly disadvise trying to 'protect' code by encrypting parts of it, it's very easy for your code to fail on >68000 if you do. What's the point anyway? Lamers will still use Action Replay to get at your code.

I never learnt anything by disassembling anyones demo. It's usually far more difficult to try and understand someone elses (uncommented) code than to write your own code from scratch.

********** 2.5a General Guidelines - ClearCacheU() ***********

CacheClearU - User callable simple cache clearing (V37)

CacheClearU()
    -636

void CacheClearU(void);

Flush out the contents of any CPU instruction and data caches.
If dirty data cache lines are present, push them to memory first.

Caches must be cleared after *any* operation that could cause
invalid or stale data. The most common cases are DMA and modifying
instructions using the processor.

Some examples of when the cache needs clearing:
      Self modifying code
      Building Jump tables
      Run-time code patches
      Relocating code for use at different addresses.
      Loading code from disk

********** 2.5b General Guidelines - CacheControl() ***********

CacheControl - Instruction & data cache control

```
oldBits = CacheControl(cacheBits,cacheMask)
  D0          -648           D0         D1

ULONG CacheControl(ULONG,ULONG);
```

This function provides global control of any instruction or data caches that may be connected to the system. All settings are global -- per task control is not provided.

The action taken by this function will depend on the type of CPU installed. This function may be patched to support external caches, or different cache architectures. In all cases the function will attempt to best emulate the provided settings. Use of this function may save state specific to the caches involved.

The list of supported settings is provided in the exec/execbase.i include file. The bits currently defined map directly to the Motorola 68030 CPU CACR register. Alternate cache solutions may patch into the Exec cache functions. Where possible, bits will be interpreted to have the same meaning on the installed cache.

IN:

cacheBits - new values for the bits specified in cacheMask.

cacheMask - a mask with ones for all bits to be changed.

OUT:

oldBits - the complete prior values for all settings.

As a side effect, this function clears all caches.

********** 2.6 General Guidelines - Hardware Access ***********

This one is aimed particularly at utility authors. I've seen some
awfully written utilities, for example (although I don't want
to single them out as there are plenty of others) the Kefrens
IFF converter.

There is NO REASON why this has to have it's own copperlist. A standard
OS-friendly version opening it's own screen works perfectly (I
still use the original SCA IFF-Converter), and multitasks properly.

If you want to write good utilities, learn C.

********** 2.7 General Guidelines - Opening Libraries Properly ***********

This has got to be one of the worst pieces of code I've
ever seen! Don't ever do this, Michel!

```
   move.l   4.w,a0          ; get execbase
   move.l   (a0),a0         ; wandering down the library list...
   move.l   (a0),a0         ; right. I think this is graphics.library

   ; now goes ahead and uses a0 as gfxbase...
```

Oh yes, graphics.library is always going to be second down the chain from
Execbase? No way!

(Note by Michel: I'm sorry... I'll never do it again :))

If you want to access gfxbase (or any other library base) OPEN the
library. Do not wander down the library chain, either by guesswork or
by manually checking for "graphics.library" in the library base name.
OpenLibrary() will do this for you.

Here is the only official way to open a library.

```
   MOVEA.L  4,a6
   LEA.L    gfxname(PC),a1
   MOVE.L   #39,d0               ; version required (here V39)
   JSR      _LVOOpenLibrary(a6)  ; resolved by linking with amiga.lib
                                 ; or by include "exec/exec_lib.i"
   TST.L d0
   BEQ.S OpenFailed
; use the base value in d0 as the a6 for calling graphics functions
; remember d0/d1/a0/a1 are scratch registers for system calls

gfxname  DC.B  'graphics.library',0
```

Don't use OldOpenLibrary! Always open libraries with a version, at least V33.
V33 is equal to Kickstart 1.2. And DON'T forget to check the result returned
in d0 (and nothing else).

OldOpenLibrary saves no cycles. All it does is

```
   moveq.l  #0,d0
   JMP      _LVOOpenLibrary(a6)
```

********** 2.8 General Guidelines - Nothing is fixed - Almost **********

Unlike inferior machines, almost everything on the Amiga can
and will change. Chipsets change, the OS changes, the processor
changes. This means you can never assume that something
that is set to a particular value on your machine will be
set to the same value on other machines. The one exception is
the pointer to EXECBASE ($0000004). This long word will always
point to the ExecBase structure, which you can use to access
every other system level function on the Amiga.

At the hardware level make sure that every hardware register
is initialised to the value you require before you start (do not
assume that bitplane modulos, for example, are always set to
zero by workbench. Under V39 OS they are not!

If you are using AGA hardware registers, make sure you check
for AGA and AGA only, AAA (when it is released) is *NOT* AGA
harware compatible.

Even the processor isn't necessarily final. It is strongly
rumoured that the Motorola MC68060 is the final member of the
68000 series, and may not even come out. Expect Amigas in 2-3
years to come with RISC chip processors running 680x0 emulation.

********** 2.9 General Guidelines - Version Numbers ***********

Put version numbers in your code. This allows the CLI version command
to determine easily the version of both your source and executable
files. Some directory utilities allow version number checking too (so
you can't accidentally copy a newer version of your source over
an older one, for example). Of course, if you pack your files the
version numbers get hidden. Leaving version numbers unpacked
was going to be added to PowerPacker, but I don't know if this is
done yet.

A version number string is in the format

```
$VER: howtocode6      7.0 (13.06.92)
^          ^          ^Version number (date is optional)
|          |
|          | File Name
|
| Identifier
```

The Version command searches for $VER and prints the string it finds
following it.

For example, adding the line to the begining of your source file

```
; $VER: MyFunDemo.s 4.0 (21.01.93)
```

and somewhere in your code

```
        dc.b        "$VER: MyFunDemo 4.0 (21.01.93)",0
```

means if you do VERSION MyFunDemo.s you will get:

MyFunDemo.s 4.0 (21.01.93)

and if you assemble and do Version MyFunDemo, you'll get

MyFunDemo 4.0 (21.01.93)

This can be very useful for those stupid demo compilations
where everything gets renamed to 1, 2, 3, etc...

Just do version 1 to get the full filename (and real date)

********** 2.10 General Guidelines - System Private ***********

System-Private

If anywhere in the manuals, includes or autodocs it says that this or that is PRIVATE or RESERVED or INTERNAL (or something similar) then

- don't read this entry
- never ever WRITE something to it
- if it's a function, then DON'T use it (*)
- don't check it for anything

Private system points can be changed without reason, or without writing it into any documentation !

(Thanks Arno!)

And to add to that, if a system structure member has a routine that allows you to alter it (for example, SetAPen() alters the Pen value in the RastPort. It is currently possible to alter the pen by poking the structure) then USE IT! Do not Poke system structures unless there is no other way to alter the value.

********** 3. Assembler **********

Choosing and Using an Assembler

1. Avoid K-Seka!
2. Problems with Devpac?
3. Problems with ArgAsm?
4. Assembling within Cygnus Ed

********** 3.1 Assembler - Avoid K-Seka! **********

It's dead and buried. Get a real assembler. Hisoft Devpac is probably the best all-round assembler, although I use ArgAsm which is astonishingly fast. The same goes for hacked versions of Seka.

Is it any coincidence that almost every piece of really bad code I see is written with Seka? No, I don't think so :-)

When buying an assembler check the following:

1. That it handles standard CBM style include files without alteration.

2. That it allows multiple sections

3. That it can create both executable and linkable code

4. 68020+ support (especially if you want to program for A1200, writing A1200 code without 68020 instructions is real stupid!)

Devpac 3.0 is probably the best all-round assembler at the moment. People on a tighter budget could do worse than look at the public domain A68K (It's much better than Seka!). I'd suggest using Cygnus Ed as your Text Editor, or Turbo Text.

********** 3.2 Assembler - Problems with Devpac? ***********

If you're using Devpac 2.x and have found that the OPT o+ flag
produces crap code, then you need to add the option o3-. This is
indeed fixed in >=V3.02

Under certain circumstances the optimiser will optimise relocate
references in the first 64Kb of your code to word addressing, which
obviously isn't very good if AmigaDOS then loads your code above
65535, which is quite likely!

o3- disables short-word address optimising.

This isn't necessary if you're creating linkable code (with l+),
and indeed may be fixed in current versions of Devpac 3

My current option setup for Devpac is:

```
   opt   l+,o+,ow+,ow1-,ow2-,ow6-,d+,CHKIMM
```

l+ sets linkable code on (as I mix C and Assember in my current projects)

o+ enables optimise mode.

ow+ enables optimiser warnings (they act as errors with SLINK, so I edit
my source when I get an optimiser warning)

ow1- disables warnings on short backwards branch optimising

ow2- disables warnings on address register indirect with displacement zero
to address register indirect optimising, again I don't want to edit my code
if I have (for example)

```
         move.l   vs_vscreen1b(a0),a1  ; vs_vscreen1b = 0
```

ow6- disables warnings if short branches forwards can be made

d+ debug information on

CHKIMM - Check Immediate values. This will report an error if any
immediate addresses are used (the most common mistake in assembler
is to leave the # from a value). Address 4 (EXECBASE) is allowed, and
other fixed addesses (eg CUSTOM - $dff000) are allowed as long as
you add a .L to the end.

```
         add.l 123,d0         ; This now gives an error!
         LEA   (CUSTOM).L,a0  ; This doesn't.
```

********** 3.3 Assembler - Problems with ArgAsm? **********

First, Argasm (unlike Devpac) from the Command Line or if called from Arexx using Cygnus Ed (my prefered system) defaults to writing linkable code, so if you want executable files you need to add

```
        opt l-    (disable linkable code)
```

If you find that your Argasm executables fail then check you haven't got any BSR's across sections! Argasm seems to allow this, but of course the code doesn't work. Jez 'Polygon' San from Argonaut software who published ArgAsm says it's not a bug, but a feature of the linker...

Yeah right Jez...

But Argasm is *fast*, and it produces non-working code *faster* than any other assembler :-)

Argonaut have abandoned ArgAsm so the last version (1.09d) is the last. There will be no more, and it doesn't support 68020+ instructions, so I've stopped using it now.

********** 3.4 Assembler - Assembling Within Cygnus Ed **********

Cygnus Ed is a wonderful system: Included in the the utils/ced directory are a few macros I wrote that may help you configure your assembler to run in CED.

To install:

1. Make sure Rexxmast is running and you have copied the macros to your REXX: directory (make a sys:rexx directory rather than assigning REXX: to S:)

2. Add the line

ced -r

to your s:user-startup. Copy the CygnusEd Activator (on Cygnus Ed V2.1x distribution disk) as C:ed, yes, that's right. Right over the top of the abysmal Commodore editor!! You lose 200Kb of fastram doing this, but believe me, it's worth it. Whenever you need to use Cygnus Ed, either type ed filename and it loads in a flash, or just press Right-ALT/Right-Shift/Return to open a new CED session.

The CygnusEd Activator is public domain, and is in the utils directory of this archive.

3. Install the commands on the keys you want to use (under the special menu).

I currently have mine set up:

F1 - devpac.ced - This calls Devpac to assemble the current file. Output is file ram:test, errors to ram:errors

F2 - argasm.ced - Same, but for Argasm

F3 - errors.ced - Open and close the error window. The error window is not editable.

F10 - ram:Test - Execute the code!

Other keys are free for C, TeX or whatever else you want to use...

********** 4. 680x0 issues **********

Using 680x0 Processors

Now the Amiga 600 is the only Amiga that uses the 68000 processor (except CDTV), many of you want to use the new 68020+ instructions to make your code, faster, better and more fun at parties!

1. How can I tell what processor I am running on?
2. How to optimise for A1200 / 68020
3. Using a 68010 processor

********** 4.1 680x0 issues - which processor am I? ***********

How can I tell what processor I am running on?

Look inside your case. Find the large rectangular (or Square) chip, read the label :-)

Or...

```
        move.l  4.w,a6
        move.w  AttnFlags(a6),d0        ; get processor flags
```

d0.w is then a bit array which contains the following bits

Bit	Meaning if set	
0	68010 processor fitted (or 68020/30/40)	
1	68020 processor fitted (or 68030/40)	
2	68030 processor fitted (or 68040)	[V37+]
3	68040 processor fitted	[V37+]
4	68881 FPU fitted (or 68882)	
5	68882 FPU fitted	[V37+]
6	68040 FPU fitted	[V37+]

The 68040 FPU bit is set when a working 68040 FPU is in the system. If this bit is set and both the 68881 and 68882 bits are not set, then the 68040 math emulation code has not been loaded and only 68040 FPU instructions are available. This bit is valid *ONLY* if the 68040 bit is set.

Don't forget to check which ROM version you're running.

DO NOT assume that the system has a >68000 if the word is non-zero! 68881 chips are available on add-on boards without any faster processor.

And don't assume that a 68000 processor means a 7Mhz 68000. It may well be a 14Mhz processor.

So, you can use this to determine whether specific processor functions are available (more on 68020 commands in a later issue), but *NOT* to determine values for timing loops. Who knows, Motorola may release a 100Mhz 68020 next year :-)

There is *NO* easy way to check for a Memory Management Unit. The MMU is present in a broken form in many 680EC30 chips.

********** 4.2 680x0 issues - 68020 Optimization ***********

A1200 speed issues:

The A1200 has a fairly large number of wait-states when accessing chip-ram. ROM is zero wait-states. Due to the slow RAM speed, it may be better to use calculations for some things that you might have used tables for on the A500.

Add-on RAM will probably be faster than chip-ram, so it is worth segmenting your game so that parts of it can go into fast-ram if available.

For good performance, it is critical that you code your important loops to execute entirely from the on-chip 256-byte cache. A straight line loop 258 bytes long will execute slower than a 254 byte one.

A loop of 258 bytes will only cause a cache miss (a word at either the beginning or the end of the loop). Only a loop of 512 bytes will cause the entire cache to miss. Of course a loop of 254 bytes will be faster than one of 258 bytes, but only marginally.

The '020 is a 32 bit chip. Longword accesses will be twice as fast when they are aligned on a long-word boundary. Aligning the entry points of routines on 32 bit boundaries can help, also. You should also make sure that the stack is always long-word aligned.

Write-accesses to chip-ram incur wait-states. However, other processor instructions can execute while results are being written to memory:

```
        move.l  d0,(a0)+        ; store x coordinate
        move.l  d1,(a0)+        ; store y coordinate
        add.l   d2,d0           ; x+=deltax
        add.l   d3,d1           ; y+=deltay

;       will be slower than:

        move.l  d0,(a0)+        ; store x coordinate
        add.l   d2,d0           ; x+=deltax
        move.l  d1,(a0)+        ; store y coordinate
        add.l   d3,d1           ; y+=deltay
```

The 68020 adds a number of enhancements to the 68000 architecture, including new addressing modes and instructions. Some of these are unconditional speedups, while others only sometimes help:

Adressing modes:

o Scaled Indexing. The 68000 addressing mode (disp,An,Dn) can have a scale factor of 2,4,or 8 applied to the data register on the 68020. This is totally free in terms of instruction length and execution time. An example is:

```
        68000                   68020
        -----                   -----
        add.w   d0,d0           move.w  (0,a1,d0.w*2),d1
        move.w  (0,a1,d0.w),d1
```

o 16 bit offsets on An+Rn modes. The 68000 only supported 8 bit

displacements when using the sum of an address register and another register as a memory address. The 68020 supports 16 bit displacements. This costs one extra cycle when the instruction is not in cache, but is free if the instruction is in cache. 32 bit displacements can also be used, but they cost 4 additional clock cycles.

- Data registers can be used as addresses. (d0) is 3 cycles slower than (a0), and it only takes 2 cycles to move a data register to an address register, but this can help in situations where there is not a free address register.

- Memory indirect addressing. These instructions can help in some circumstances when there are not any free register to load a pointer into. Otherwise, they lose.

New instructions:

- Extended precision divide an multiply instructions. The 68020 can perform 32x32->32, 32x32->64 multiplication and 32/32 and 64/32 division. These are significantly faster than the multi-precision operations which are required on the 68000.

- EXTB. Sign extend byte to longword. Faster than the equivalent EXT.W EXT.L sequence on the 68000.

- Compare immediate and TST work in program-counter relative mode on the 68020.

- Bit field instructions. BFINS inserts a bitfield, and is faster than 2 MOVEs plus and AND and an OR. This instruction can be used nicely in fill routines or text plotting. BFEXTU/BFEXTS can extract and optionally sign-extend a bitfield on an arbitrary boundary. BFFFO can find the highest order bit set in a field. BFSET, BFCHG, and BFCLR can set, complement, or clear up to 32 bits at arbitrary boundaries.

- On the 020, all shift instructions execute in the same amount of time, regardless of how many bits are shifted. Note that ASL and ASR are slower than LSL and LSR. The break-even point on ADD Dn,Dn versus LSL is at two shifts.

- Many tradeoffs on the 020 are different than the 68000.

- The 020 has PACK an UNPACK which can be useful.

********** 4.3 680x0 issues - Using a 68010 processor ***********

Using a 68010 processor

The 68010 is a direct replacement for the 68000 chip, it can
be fitted to the Amiga 500,500+,1500, CDTV and 2000 without any
other alterations (I have been told it will not fit an A600).

The main benefit of the 68010 over the 68000 is the loop cache mode.
Common 3 word loops like:

```
          moveq  #50,d0
.lp       move.b (a0)+,(a1)+  ; one word
          dbra   d0,.lp       ; two words
```

are recognised as loops and speed up dramatically on 68010.

********** 5. Action replay **********

Action Replay Cartridges

1 Entering 'sysop mode'
2 How Action Replay 2 works

********** 5.1 Action replay - Entering 'Sysop Mode' **********

Action Replay is great fun, even more so if you get into the
'sysop mode' (Allows disassembly of ram areas not previously
allowed by Action Replay, including non-autoconfig ram and
the cartridge rom!)

To get into sysop mode on Action Replay 1 type:

LORD OLAF

To get into sysop mode on Action Replay 2 type:

MAY
THE
FORCE
BE
WITH
YOU

To get into sysop mode on Action Replay 3 type the same as
Action Replay 2. After this you get a message
"Try a new one".
Then type in

NEW

and sysop powers are granted!

********** 5.2 Action replay - How Action Replay 2 works **********

The Action Replay 2, How it and the Amiga works, and why

For all the hackers amongst you lot, especially those with one of Datel's excellent Action Replay Mk.][s (unlike the first one, which was useless), here is a little technical info about it, and how to protect against it.

********** 5.2a Action replay - The Cartridge internals **********

The cart's internals

The cartridge contains 128k of Rom (contained in two 27512 (64kx8bits) roms) and 32k of static ram. All the chips are surface-mount types, and are buggers to get off intact with a soldering iron. My advice is, don't bother! (There is also a scattering of custom PLAs and things, but they're not very interesting.) The Rom is addressable from $400000-$41ffff (and is repeated due to partial address decoding from $420000-$43ffff). The Ram is from $440000-$447fff, and is repeated loads of times for the same reason.

Normally, this Rom & Ram is totally invisible & undetectable. It is switched out of the address space, and there is *nothing* you can do to switch it in via software (there is an exception to this, see later).

Pressing the Button..

When you press the button

When you press the button, the cartridge generates a level 7 (non-maskable) interrupt, and then when the 68000 comes to get the vector (at $7c) for the interrupt, the cartridge whangs on the OVR* line on the expansion port, which has the effect of disabling all the amiga's internal chip memory!

The cart then makes its internal Rom appear instead of the chip ram from $00000-$80000. This Rom (and its Ram) simultaneously appears at their normal addresses at $400000/$440000. If this sounds like magic, it's actually incredibly simple to do in hardware!

What happens, and why..

********** 5.2c Action replay - what happens, and why ***********

A Quick explanation of what happens and why

As a matter of interest, the cart is doing something very similar to what the Amiga normally does on a reset, which is to switch all of Kickstart in from $00000-$80000, thereby providing the reset vector. That's why if you do a RESET instruction, then your code will most likely blow up instantly as all chip ram disappears! (The way to make thing return to normal is to set the CIA OVL* (OVerLay) signal back to 0, but if your code is dead, this is easier said than done.. however it can be done by some deeply sneaky programming.. see the Double Dragon][/Shinobi protection that I coded a while back)

Anyway, I digress. This new level 7 vector from the cartridge is $400088, a routine in the Rom proper which first restores the chip ram,etc to normal. Then it treks off into the bowels of the cartridge software.. you can follow it through and ReSource it if you like!

This is, however, only one of the ways to get into the cartridge. There are two others..

Two ways to get into the cart without pressing the button

********** 5.2d Action replay - two ways to get into the cart **********

Two ways to get into the cart without pressing the button

(Or: 'Futility- an object lesson'!)

1)
This first, is the way the cart boots up on a reset (displaying that little piccy). The way this works is indecently sneaky! Instead of (as I had expected) either intercepting the reset wholesale, or appearing to the Kickstart reset code as an autoboot cartridge (which can be done by making your Rom appear at $F00000 on reset with $1111 as the first word - see $fc00d2 in the Rom), the AR2 goes for a MUCH sneakier route!

It detects processor accesses to location $0000008, and when one happens, it effectively 'presses' it's own button. (i.e. does the level 7 business) Now, it just so happens that the Kickstart Rom does a MOVE to location $8 very early on, when it sets up the exception vectors, so voila! There it goes into the cart, returning later when it feels like it! At this point, I feel I should deeply disappoint those of you with a more technical bent (oo-er!)... You may be thinking that if you use the processor to generate a reset (with the RESET instruction), and then access location $8, then the cart will reveal it's presence, and thereby you can protect against it!

No way jose!

The designer of the cart was clever enough to put in a circuit that can actually tell the difference between you plonking your fingers on the 3-keys of doom, and the processor doing a RESET.. Not easy to do! (It actually times the reset pulse, and whereas the processor RESET instruction makes a really short one, the keyboard reset lasts for about ½ a second, and only responds to a location 8 access after a genuine reset.)

Smart eh?

2)
The second way is a bit simpler. When you set a breakpoint with the cart, or set the exceptions with SETEXCEPT, what the cart does is to put a set of TST.B $BFE001 instructions at location $100, with the vectors in question pointing there. What then happens, is then when you get an exception, the TST.B $BFE001 is executed, the cart detects it happening, and does the Level 7 thang. Now. This TST.B can't just be anywhere. It has to be from $100 to $120ish.

"Ah HA!", you think,"I'll just have a few 'TST.B $BFE001's executed at $100 before my game loads!!"

....Oh no you won't! This whole thing is only enabled after the user does a SETEXCEPT or sets up a breakpoint! Normally the above has NO EFFECT AT ALL! Who's a clever little cart designer then???

How does it know what's going on with the custom chip registers?

********** 5.2e Action replay - How does it know what's going on? ***********

How does it know what's going on with the custom chip registers?

I knew you would ask me that one! Ok, get ready for some interesting (?) technical info.

As you may well know, all the custom chip registers at $DFF000-$DFF200 are EITHER read-only OR write-only, never, ever, both. This is for a good design reason (take my word for it). Some of the more interesting registers have separate read registers (e.g. DMACONR) so you can tell what the register itself contains. Most don't...

Oh dear!

It is therefore simply NOT POSSIBLE to tell what was last written to, say, COLOUR00 ($DFF180), without either; A) Asking the person looking at the screen to describe the border colour, or B) Extra hardware.

Not entirely suprisingly, the AR2 goes for the latter solution. First, a quickie lesson in how the amiga's custom chips communicate internally...

The custom chip 'registers' are not really actual memory at all. What that area ($Dff000-$dff200) actually is is a 'window' into the internal custom-chip computer system. This system consists of a 'bus' (i.e. a channel for data consisting of several physical connections grouped together, each one carrying one bit of the data that it being passed) that is connected to all the custom chips inside the Amiga. (See the pins labeled 'RGA1-8' which contains the number $000-$200 of the custom register, and 'D0-D15' which contain the 16 bits of data being transferred, that are on all of the main custom chips)

Using this, all the various registers (which in physical reality are located scattered about in various of the custom chips, according to their usage. I.e. DIWSTRT is in Denise, which generates the display, and BLTSIZE is in Fat Agnus, which contains the blitter amongst other things) can be read or written.

These registers do not exist solely for the purpose of being read/written by the 68000. Certainly most of them have no purpose than to be set up by the main processor in order to tell the appropriate custom chip what to do, but others are really of no use to the programmer whatsoever. For example, the xxxxxxDAT registers (more on them later).

Apart from the 68000 using this bus to communicate with the custom chip registers, the chips themselves use it all the time too!

For example, how the screen display is generated...

Fat Agnus does all the DMA handling; i.e. it is the chip that, when a bit of data (screen,disk,audio,etc) has to be transferred to/from chip memory, actually does the donkey work of read/writing the value to the appropriate main memory location. (Main memory is a separate system to this internal custom chip world, and Agnus is the interface)

For screen DMA, the data actually is destined for the Denise chip, which is a separate lump of silicon, so how does it get from Agnus (that has just fetched it from Ram) to Denise? Via this internal bus! This is where those registers

that no-one seems to know about come in. I mean the xxxxDAT registers!

BPL1DAT (for example) is not really meant to be accessed by the user at all! It is there for internal usage.. i.e. Agnus reads a word of the screen memory from the outside world, and writes it into BPL1DAT, which is a physical register inside the Denise chip.

Now, this operation is functionally the same as you doing a MOVE.W into BPL1DAT, but it is done by Agnus, with no help from the processor. This is DMA. If you read from VHPOSR, then the read request from the 68000 would be passed to Agnus, which would then consult the internal bus, and then deliver the value back.

Ok, basically the processor is just a spectator on this internal register bus. (Am I being too patronising? Sorry.)

Right, so now you know that the custom chip registers are not really part of main memory at all (that's why the Copper can only work within this small world of registers and not with all of Ram), and you know/already knew that there are many registers that you cannot read at all.

Back on the actual subject in hand.. i.e. how the AR2 knows what these internal registers contain, when it is simply not possible to read them....!

Answer: It doesn't.

What it actually does, is use an idea stolen from Romantic Robot (a company that made the first decent freezer cart for the Amstrad CPC, which also had write-only registers) which is to make a bit of sneaky hardware that effectively sits there and watches the 68000 like a hawk. Whenever the 68000 does a write operation to a chip register, it makes an internal copy of the value being written! What this amounts to is $200 bytes of totally invisible (invisible, that is, until you whack the button, when it appears - to be read - in another area of memory) Ram that contains all the values most recently written by the processor to any custom chip register.

Handy eh? So the action replay has all those values you wrote to COLOUR00, DSKSYNC,etc,etc copied down in its' own bit of ram!

Now, here's where the major catch comes...

You may have noticed two things....
A) Only the value last written is kept, and
B) Only VALUES WRITTEN BY THE PROCESSOR can be kept.

The first argument is not really that important, but the second one IS!

How to protect against Action Replay Mk.][

********** 5.2f Action replay - How to protect against AR mk 2 ***********

3 example ways to protect against Action Replay Mk.][

A)
There is NO POSSIBLE WAY that the AR2 can EVER TELL WHAT THE COPPER HAS WRITTEN TO A CUSTOM CHIP REGISTER.

A fundimental flaw, and one which it is not possible to solve in hardware without having a set of leads connected inside your amiga. The chip register bus is not available on the explansion port.

"Hang on!", I hear you cry (?),"how come it knew about that DIWSTRT that I only wrote in my copper list?"

Ah ha. Well, this is where the sneaky programming in the cart comes in. It knows where your copper list is (after all, you must have written to COP1LC at some point.. with the processor) and so what it does is to follow your copper list through and it actually updates its little internal copy of the custom registers to do the changes that the copper must have done!

I.e. if you did a CLR $DFF180 to clear colour0, and then in your copper list you had a MOVE $FFFF,$0180 to there, the AR2 would only have your processor CLR stored by the hardware, but the software would see the copper instruction and alter its record. In fact, it has no way of actually knowing whether this copper instruction was ever run or not, it just guesses!

So, the main form of protection is to set up all the important chip registers using a temporary one-off copper list, one that, once it has run, is dumped and a new copper list (the proper in-game one) is started instead.

The AR2 will never know about any of the chip writes done in the first copper list, if you press the button when the final one is running.

This is better than it seems, because the AR2 changes loads of things when it runs it's own monitor (i.e. resolution, # of bitplanes, pointers,etc,etc) and relies on knowing what to put back into the registers when it returns to your code. If it has a totally false idea of what was in, for example, BLTCON0, or DSKSYNC, then there is no way your program will ever run again!

That's the only practical way to protect against the cart, and as you can see, it is naff in that it doesn't either a) detect the cart before the program runs, or b) stop you going into the monitor.

The other ways are....
("The Dog-In-A-Manger Approch".. if I can't.. neither can you!)

B)
Point the Supervisor stack to an ODD address, and run your program in user mode, with NO INTERRUPTS! When you get an interrupt, the processor always enters supervisor mode, switches over to the supervisor stack, and pushes on the address to return to after the interrupt and the current Status Register value. If the address that it tries to push these to is odd...? Kapooof. Not just an address error, but the address error itself also tries to push words onto the odd-stack, and you get a double-exception.. i.e. total 68000

lock-up. It will not recover until you do a hardware reset.

This is absolutely the best way to fuck ANY cart up. Press the button when this has been done and the entire computer crashes totally. But... Can you write a game without using interrupts?

(The 'Say kids.. what time is it?' approach)

C) Use the CIA Time-of-day alarm. This is semi-complex..

Each of the 2 CIAs have 'Time of Day' clocks. These are clocks that run on the conventional hour/minute/second scale, and are driven by the system clock. They have to be set to the right time after every reset, so are almost never used for their intended purpose. Thing is, the clocks also have an alarm facility, whereby you can get an interrupt from the CIA when the current time=the preset alarm time. There are 3 registers (hours/minutes/seconds) that if read, contain the current time, if written in mode 1 (mode is set by a register bit), will change the current time, and if written in mode 2, will change the alarm time.

This alarm time cannot ever be read. So.. what you do is...

Set your alarm time for, say, 00:00:10, then set the current time to 00:00:00, and enable the interrupt. Start your game going. When you get the alarm interrupt, set the current time back to 00:00:00, and in another 10 seconds you will get another interrupt, and so on. If, however, you notice that the time has ever gone past 00:00:10 without you getting an interrupt, or that the alarm occurs at the wrong time, then you know that someone has tampered with the program and didn't set the right alarm time! I know it sounds complicated, but if you use a weird alarm time, then noone will ever know what the correct value to set it to is, and so the 'freezer' can never produce a copy of the game that will unfreeze and work.

Both these approches use features of the Amiga/68000 that cannot be got around. The first one can be fixed by having loads of internal connections into the Amiga, but no-one will want to do that. Unfortunately, at time of writing (31st Jan'91), I can't think of another way to stop the cart getting into the monitor. Like I said, it's quite well designed!

If you can find a bug in the software to exploit, all well and good, but remember bugs can be fixed!

P.P.P.P.S. Thanx to Bob & Jim for the help.

Brought to you by
GREMLIN of MAYHEM (finished 5:50am 31st Jan 1991)

The AGA Chipset (Amiga 1200/4000)

**** WARNING ****

AGA Registers are temporary. They will change. Do not rely on this documentation. No programs written with this information can be officially endorsed or supported by Commodore. If this bothers you then stop reading now.

Future Amigas will *NOT* support *ANY* of the new AGA registers. If you want your product to work on the next generation of Amigas then either:

a) Program for ECS only (*MOST* ECS will be supported. Don't rely on Productivity or SuperHires mode via hardware though!)

b) Program your displays via the OS, either using graphics.library (views) or intuition.library (screens). If you use the OS then any UserCopperList code you add must *ONLY* be for ECS level instructions or lower (so, sorry, no 24-bit rainbows).

I have decided to include this material as there are still reasons for people to program AGA hardware, especially demo coders. PLEASE do not let me down by using this information to write commercial software. If this happens I will have to remove the AGA docs from HowToCode.

And as soon as Commodore provide a suitable way for all programmers to access the power of the A1200/4000 chipset in a supportable way, this file will be removed and replaced with one that tells you how to code properly. But in the meantime......

********** 6.1 Aga - What chipset am I using? ***********

How do I tell what chipset I am using?

Do *NOT* check library revision numbers, V39 OS can and does
run on standard & ECS chipset machines (My Amiga 3000
is currently running V39).

This code is a much better check for AGA than in howtocode4!!!!!

```
GFXB_AA_ALICE   equ 2
gb_ChipRevBits0 equ $ec

; Call with a6 containing GfxBase from opened graphics.library

   btst  #GFXB_AA_ALICE,gb_ChipRevBits0(a6)
   bne.s is_aa
```

This will not work unless the V39 SetPatch command has been
executed. If you *must* use trackloader demos then
execute the graphics.library function

```
    SetChipRev(chipset)
```

This is a V39 function (No Kickstart 3.0? Then you haven't got
AGA!).

You can set the chipset you require with the following parameters:

```
   Normal = $00
   ECS    = $03       (Only on ECS or higher)
   AGA    = $0f       (Only on AGA chipset machines)
   Best   = $ffffffff (This gives best possible on machine)
```

This is called in the system by SetPatch.

The code in howtocode4 also had major problems when being
run on non ECS machines (without Super Denise or Lisa), as the
register was undefined under the original (A) chipset, and
would return garbage, sometimes triggering a false AGA-present
response.

********** 6.2 Aga - Programming the AGA hardware **********

Programming AGA hardware

********** 6.2a Aga - bitplanes ***********

Bitplanes:

Set bits 0 to 7 bitplanes as before in BPLCON0 (for 0 to 7 bitplanes)

For 8 bitplanes you should set bit 4 (BPU3) of BPLCON0
bits 12 to 14 (BPU0 to BPU2) should be zero.

Using 64-colour mode (NOT extra halfbrite) requires setting the
KILLEHB (bit 9) in BPLCON2.

Super Hires can be enabled by bit 6 (SHRES) of BPLCON0

********** 6.2b Aga - Colour Registers ***********

Colour Registers

There are now 256 24-bit colour registers, all accessed through the original 32 12-bit colour registers. If you suspect this sounds like it could be messy, then you're right, it is!

AGA works with 8 differents palettes of 32 colors each, re-using colour registers from COLOR00 to COLOR31

You can choose the palette you want to access via bits 13 to 15 of register BPLCON3.

BANK2 bit 15	BANK1 bit 14	BANK0 bit 13	Selected palette
0	0	0	Palette 0 (color 0 to 31)
0	0	1	Palette 1 (color 32 to 63)
0	1	0	Palette 2 (color 64 to 95)
0	1	1	Palette 3 (color 96 to 127)
1	0	0	Palette 4 (color 128 to 159)
1	0	1	Palette 5 (color 160 to 191)
1	1	0	Palette 6 (color 192 to 223)
1	1	1	Palette 7 (color 224 to 255)

To move a 24-bit colour value into a colour register requires two writes to the register:

First clear bit 9 (LOCT) of BPLCON3
Move high nibbles of each colour component to colour registers

Then set bit 9 (LOCT) of BPLCON3
Move low nibbles of each colour components to colour registers

For example, to change colour zero to the colour $123456

```
   lea      (CUSTOM.L),a0
   move.w   #$0135,COLOR00(a0)
   move.w   #$0200,BPLCON3(a0)
   move.w   #$0245,COLOR00(a0)
   move.w   #$0000,BPLCON3(a0)
```

********** 6.2c Aga - AGA sprites ***********

Sprites

To change the resolution of the sprite, just use bit 7 and 6 of register BPLCON3

bit 7	bit 6	Resolution
0	0	ECS Defaults (Lo-res/Hi-res = 140ns, Superhires = 70ns)
0	1	Always lowres (140ns)
0	1	Always hireres (70ns)
1	1	Always superhires (35ns)

For 32-bit and 64-bit wide sprites use bit 3 and 2 of register FMODE ($dff1fc) Sprite format (in particular the control words) vary for each width.

bit 3	bit 2	Wide	Control Words
0	0	16 pixels	2 words (normal)
1	0	32 pixels	2 longwords
0	1	32 pixels	2 longwords
1	1	64 pixels	2 double long words (4 longwords)

Wider sprites are not available under all conditions.

It is possible to choose the color palette of the sprite.
This is done with bits 0 to 3 (even) and 4 to 7 (odd) of register $010C.

bit 3 bit 7	bit 2 bit 6	bit 1 bit 5	bit 0 bit 4	Even sprites Odd Sprites
0	0	0	0	$0180/palette 0 (coulor 0)
0	0	0	1	$01A0/palette 0 (color 15)
0	0	1	0	$0180/palette 1 (color 31)
0	0	1	1	$01A0/palette 1 (color 47)
0	1	0	0	$0180/palette 2 (color 63)
0	1	0	1	$01A0/palette 2 (color 79)
0	1	1	0	$0180/palette 3 (color 95)
0	1	1	1	$01A0/palette 3 (color 111)
1	0	0	0	$0180/palette 4 (color 127)
1	0	0	1	$01A0/palette 4 (color 143)
1	0	1	0	$0180/palette 5 (color 159)
1	0	1	1	$01A0/palette 5 (color 175)
1	1	0	0	$0180/palette 6 (color 191)
1	1	0	1	$01A0/palette 6 (color 207)
1	1	1	0	$0180/palette 7 (color 223)
1	1	1	1	$01A0/palette 7 (color 239)

See Resetting sprites for an OS legal method of setting sprite resolution.

Alignment Restrictions

Bitplanes, sprites and copperlists must be, under certain circumstances, 64-bit aligned under AGA. Again to benefit from maximum bandwitdh bitplanes should also only be multiples of 64-bits wide, so if you want an extra area on the side of your screen for smooth blitter scrolling it must be *8 bytes* wide, not two as normal.

This also raises another problem. You can no longer use AllocMem() to allocate bitplane/sprite memory directly.

Either use AllocMem(sizeofplanes+8) and calculate how many bytes you have to skip at the front to give 64-bit alignment (remember this assumes either you allocate each bitplane individually or make sure the bitplane size is also an exact multiple of 64-bits), or you can use the new V39 function AllocBitMap() .

********** 6.2e Aga - The Magic FMode Reigster ***********

The Magic FMode Register

If you set your 1200/4000 to a hiresmode (such as 1280x512 Superhires 256 colours) and disassemble the copperlist, you find fun things happen to the FMODE register ($dff1fc). The FMODE register determines the amount of words transferred between chipram and the Lisa chip in each data fetch (I think)....

$dff1fc bits 0 and 1 value

$00 - Normal (word aligned bitmaps) - for standard ECS modes and up to 8 bitplanes 320x256

$01 - Double (longword aligned bitmaps) - for 640x256 modes in more than 16 colours

$10 - Double (longword aligned bitmaps) - Same effect, for 640x256 modes but different things happen... Not sure why!

$11 - Quadruple [x4] (64-bit aligned bitmaps) - for 1280x256 modes...

Fetch Modes Required for Displays [table]

********** 6.2f Aga - Fetch Modes for displays ***********

Fetch Mode Required for Displays

ALL ECS and lower screenmodes require only 1x datafetch. All modes run *FASTER* with at least 2x bandwidth, so try and use 2x bandwitdh if possible.

	Planes	Colours	Fetchmode
LORES (320x256)			
	6	64	1
	7	128	1
	8	256	1
	8	HAM-8	1
HIRES (640x256)	5	32	2
	6	64	2
	7	128	2
	8	256	2
	8	HAM-8	2
SUPER-HIRES	1	2	1
(1280x256)	2	4	1
	3	8	2
	4	16	2
	5	32	4
	6	64	4
	7	128	4
	8	256	4
	8	HAM-8	4
PRODUCTIVITY	1	2	1
(640x480,etc)	2	4	1
	3	8	2
	4	16	2
	5	32	4
	6	64	4
	7	128	4
	8	256	4
	8	HAM-8	4

This table only shows the minimum required fetchmode for each screen. You should always try and set the fetchmode as high as possible (if you are 64-bit aligned and wide, then $11, if 32-bit aligned and wide $01, etc...)

Bits 2 and 3 do the same for sprite width, as has been mentioned elsewhere...

Remember... To take advantage of the increased fetchmodes (which give you more processor time to play with!) your bitmaps must be on 64-bit boundaries and be multiples of 64-bits wide (8 bytes)

Smoother Hardware Scrolling

Extra bits have been added to BPLCON1 to allow smoother hardware scrolling and scrolling over a larger area.

Bits 8 (PF1H0) and 9 (PF1H1) are the new hi-resolution scroll bits for playfield 0 and bits 12 (PF2H0) and 13 (PF2H1) are the new bits for playfield 1.

Another two bits have been added for each bitplane at bits 10 (PF1H6) and 11 (PF1H7) for playfield 1 and bits 14 (PF2H6) and 15 (PF2H7) to increase the maximum scroll range from 16 lo-res pixels to 64 lo-res pixels (or 256 superhires pixels).

Normal 0-16 positions therefore are normal, but it you want to position your screen at a (super) hires position you need to set the new bits, or if you require smooth hardware scrolling with either 2x or 4x Fetch Mode .

********** 6.2h Aga - What is the HAM-8 Format? ***********

What is HAM-8 Format?

Ham8 mode is enabled when the HAM bit is set in BPLCON0 and 8 bitplanes
are specified.

Ham-8 uses *lower* two bits as the command (either
new register (%00), or alter Red, Green or Blue component, as in
standard HAM), and the *upper* 6 bits (planes 2 to 7) as the
register(0 to 63), or as an 6 bit hold-and-modify value to modify
the top 6 bits of an 8-bit colour component.

The lower two bits of the colour component are not altered, so
initial palettes have to be chosen carefuly (or use Art
Department Professional or anything that selects colours better)

********** 6.3 Aga - Monitor type problems **********

Monitor Problems

Unfortunately the A1200/AGA chipset does not have the deinterlacer circuitry present in the Amiga 3000, but instead has new 'deinterlaced' modes. This gives the A1200 the capability of running workbench (and almost all OS legal software) the ability to run flicker free at high resolution on a multiscan or Super VGA monitor.

Unlike the Amiga 3000 hardware it produces these flicker free modes by generating a custom copperlist, so any programs that generate their own copperlists will continue to run at the old flickery 15Khz frequency unless they add their own deinterlace code.

This is a big problem for many A1200 owners as there are very few multiscan monitors that support 15Khz displays now. Most multiscan monitors will not display screen at less than 27Khz. People with A1200/4000 and this kind of monitor *CANNOT* view any games or demos that write their own copperlists.

Can you help them out? Unfortunately it's not easy. Deinterlacing is done in AGA by doing two things.

Firstly different horizontal and vertical frequencies are set (These are set to unusual values for anyone used to Amiga or PC displays! For example, DblPal is set by default to 27Khz horizontal and 48Hz vertical) It's important to realise that the vertical frequency changes too!

Seondly, for non-interlaced screens, bitplane scandoubling is enabled (bit BSCAN2 in FMODE) This repeats each scanline twice. A side effect of this is that the bitplane modulos are unavailable for user control.

So... There are three options.

1. Write nasty copperlist code to work with both standard and promoted displays (Not a good idea!)

2. Use the OS and set up your displays legally, asking the Display Database for a screenmode that is available for the current monitor.

3. Give up, and say your demo requires a 15Khz monitor.

I think most people will go for option 3. The Commodore 1084/1085, Phillips 8833/8852 and the Commodore 1950/1960/1940/1942 monitors are all capable of running 15Khz screens.

********** 6.3a Aga - AllocMem() **********

AllocMem -- allocate memory given certain requirements

```
memoryBlock = AllocMem(byteSize, attributes)
    D0            -198      D0         D1

void *AllocMem(ULONG, ULONG);
```

********** 6.3b Aga - AllocBitMap() **********

AllocBitMap -- Allocate a bitmap and attach bitplanes to it. (V39)

```
bitmap=AllocBitMap(sizex,sizey,depth, flags, friend_bitmap)
          -918       d0    d1    d2     d3       a0

struct BitMap *AllocBitMap(ULONG,ULONG,ULONG,ULONG, struct BitMap *);
```

Allocates and initializes a bitmap structure. Allocates and initializes bitplane data, and sets the bitmap's planes to point to it.

IN:

sizex = the width (in pixels) for the bitmap data.

sizey = the height (in pixels).

depth = the number of bitplanes deep for the allocation.

flags = BMF_CLEAR - Clear the bitmap.

BMF_DISPLAYABLE - bitmap displayable on AGA machines in all modes.

BMF_INTERLEAVED - bitplanes are interleaved

friend_bitmap = pointer to another bitmap, or NULL. If this pointer If present, bitmap will be allocated so blitting between the two is simplified.

SEE ALSO

FreeBitMap()

********** 6.3c Aga - FreeBitMap() **********

FreeBitMap -- free a bitmap created by AllocBitMap

```
FreeBitMap(bm)
  -924     a0
```

VOID FreeBitMap(struct BitMap *)

Frees bitmap and all associated bitplanes

IN:
   bm = A pointer to a BitMap.

********** 6.4 Aga - Resetting AGA sprite resolution ***********

This is a totally OS-legal way of resetting sprite resolution to 140ns
(ECS default). call FixSpritesSetup: *BEFORE* your LoadView(NULL) in your
startup code, and ReturnSpritesToNormal: *BEFORE* the LoadView(wbview)
that returns workbench in your exit code:

Here is the assembler version of the code: See startup.asm
for an integrated example of this code:

```
; Setup code - assumes V39 Kickstart or higher


FixSpritesSetup:
        move.l   _IntuitionBase,a6          ; open intuition.library first!
        lea      wbname,a0
        jsr      _LVOLockPubScreen(a6)

        tst.l    d0                         ; Could I lock Workbench?
        beq.s    .error                     ; if not, error
        move.l   d0,wbscreen
        move.l   d0,a0

        move.l   sc_ViewPort+vp_ColorMap(a0),a0
        lea      taglist,a1
        move.l   _GfxBase,a6                ; open graphics.library first!
        jsr      _LVOVideoControl(a6)       ;

        move.l   resolution,oldres          ; store old resolution

        move.l   #SPRITERESN_140NS,resolution
        move.l   #VTAG_SPRITERESN_SET,taglist

        move.l   wbscreen,a0
        move.l   sc_ViewPort+vp_ColorMap(a0),a0
        lea      taglist,a1
        jsr      _LVOVideoControl(a6)       ; set sprites to lores

        move.l   wbscreen,a0
        move.l   _IntuitionBase,a6
        jsr      _LVOMakeScreen(a6)
        jsr      _LVORethinkDisplay(a6)     ; and rebuild system copperlists

; Sprites are now set back to 140ns in a system friendly manner!

.error
        rts

ReturnSpritesToNormal:
; If you mess with sprite resolution you must return resolution
; back to workbench standard on return! This code will do that...

        move.l   wbscreen,d0
        beq.s    .error
        move.l   d0,a0

        move.l   oldres,resolution          ; change taglist
        lea      taglist,a1
        move.l   sc_ViewPort+vp_ColorMap(a0),a0
```

```
        move.l   _GfxBase,a6
        jsr      _LVOVideoControl(a6)       ; return sprites to normal.

        move.l   _IntuitionBase,a6
        move.l   wbscreen,a0
        jsr      _LVOMakeScreen(a6)         ; and rebuild screen

        move.l   wbscreen,a1
        sub.l    a0,a0
        jsr      _LVOUnlockPubScreen(a6)

.error
        rts


oldres          dc.l  0
wbscreen        dc.l  0

taglist         dc.l  VTAG_SPRITERESN_GET
resolution      dc.l  SPRITERESN_ECS
                dc.l  TAG_DONE,0

wbname          dc.b  "Workbench",0
```

```
********** 6.4a Aga - LockPubScreen() ***********

LockPubScreen -- Put a lock on a Public Screen.

screen = LockPubScreen( Name )
 D0           -510          A0

struct Screen *LockPubScreen( UBYTE * );

Prevents a public screen (or the Workbench) from closing.
```

********** 6.4b Aga - UnlockPubScreen() ***********

UnlockPubScreen -- Remove lock from a Public Screen.

```
UnlockPubScreen( Name, [Screen] )
     -516          A0    A1

VOID UnlockPubScreen( UBYTE *, struct Screen * );
```

Releases a lock from LockPubScreen()

IN:
   Usually Name = NULL and Screen = pointer returned by LockPubScreen()

********** 6.4c Aga - VideoControl() **********

VideoControl -- Parse tags on viewport colormap.

```
err = VideoControl( cmap , tags )
d0         -708      a0    a1

ULONG VideoControl( struct ColorMap *, struct TagItem * );
```

Process the tag commands on the colormap.

IN:
   cm   = pointer to struct ColorMap
   tags = pointer to a table of videocontrol tagitems.

OUT:
   error = NULL if no error occured.

********** 6.4d Aga - SetChipRev() ***********

SetChipRev -- Enables Chip Set features

```
chipbits = SetChipRev(Rev)
   -888               d0
```

IN:
   Rev - Revision to be enabled ($ffffffff for best possible)

OUT:
   chipbits - State of chipset on exit.

Only call this routine once. It is called by the OS in SetPatch, but you should use it if you are writing Non-DOS demos or games.

********** 7. Blitter ***********

Using the Blitter

********** 7.1 Blitter - OwnBlitter()/DisownBlitter() **********

If you are using the blitter in your code and you are leaving the system intact (as you should) always use the graphics.library functions OwnBlitter() and DisownBlitter() to take control of the blitter. Remember to free it for system use, many system functions (including floppy disk data decoding) use the blitter.

OwnBlitter() does not trash any registers. I guess DisownBlitter() doesn't either, although Chris may well correct me on this, and they are fast enough to use around your blitter code, so don't just OwnBlitter() at the beginning of your code and DisownBlitter() at the end, only OwnBlitter() when you need to.

********** 7.1a Blitter - OwnBlitter() ***********

OwnBlitter -- get the blitter for private usage

OwnBlitter()
   -456

void OwnBlitter(void);

If blitter is available return immediately with the blitter
locked for your exclusive use. If the blitter is not available
put task to sleep. It will be awakened as soon as the blitter
is available. When the task first owns the blitter the blitter
may still be finishing up a blit for the previous owner. You
must do a WaitBlit before actually using the blitter registers.

Calls to OwnBlitter() do not nest. If a task that owns the
blitter calls OwnBlitter() again, a lockup will result.
(Same situation if the task calls a system function
that tries to own the blitter).

********** 7.1b Blitter - DisownBlitter() **********

DisownBlitter - return blitter to free state.

```
DisownBlitter()
    -462
```

```
void DisownBlitter( void );
```

Free blitter up for use by other blitter users.

********** 7.1c Blitter - QBlit() ***********

QBlit -- Queue up a request for blitter usage

```
QBlit( bp )
-276   a1

void QBlit( struct bltnode * );
```

Link a request for the use of the blitter to the end of the current blitter queue. The pointer bp points to a blit structure containing, among other things, the link information, and the address of your routine which is to be called when the blitter queue finally gets around to this specific request. When your routine is called, you are in control of the blitter ... it is not busy with anyone else's requests. This means that you can directly specify the register contents and start the blitter. Your code must be written to run either in supervisor or user mode on the 68000.

IN:
bp - pointer to a blit structure

Your routine is called when the blitter is ready for you. In general requests for blitter usage through this channel are put in front of those who use the blitter via OwnBlitter and DisownBlitter. However for small blits there is more overhead using the queuer than Own/Disown Blitter.

********** 7.1d Blitter - QBSBlit() **********

QBSBlit -- Synchronize the blitter request with the video beam.

```
QBSBlit( bsp )
 -294    a1

void QBSBlit( struct bltnode * );
```

Call a user routine for use of the blitter, enqueued separately from the queue. Calls the user routine contained in the blit structure when the video beam is located at a specified position onscreen. Useful when you are trying to blit into a visible part of the screen and wish to perform the data move while the beam is not trying to display that same area. (prevents showing part of an old display and part of a new display simultaneously). Blitter requests on the QBSBlit queue take precedence over those on the regular blitter queue. The beam position is specified the blitnode.

IN:
   bsp - pointer to a blit structure.

********** 7.1e Blitter - WaitBlit() **********

WaitBlit -- Wait for the blitter to finish.

```
WaitBlit()
 -228
```

********** 7.2 Blitter - Blitter Timing ***********

Blitter timing

Another common cause for demo crashes is blitter timing.

Assuming that a particular routine will be slow enough that a blitter wait is not needed is silly. Always check for blitter finished, and wait if you need to.

Don't assume the blitter will always run at the same speed too. Think about how your code would run if the processor or blitter were running at 100 times the current speed. As long as you keep this in mind, you'll be in a better frame of mind for writing code that works on different Amigas.

Another big source of blitter problems is using the blitter in interrupts.

Many demos do all processing in the interrupt, with only a

```
.wt   btst     #6,$bfe001  ; is left mouse button clicked?
      bne.s    .wt
```

loop outside of the interrupt. However, some demos do stuff outside the interrupt too. Warning. If you use blitter in both your interrupt and your main code, (or for that matter if you use the blitter via the copper and also in your main code), you may have big problems....

Take this for example:

```
   lea      $dff000,a5
   move.l   GfxBase,a6
   jsr      _LVOWaitBlit(a6)
   move.l   #-1,BLTAFWM(a5)                ; set FWM and LWM in one go
   move.l   #source,BLTAPT(a5)
   move.l   #dest,BLTDPT(a5)
   move.w   #%100111110000,BLTCON0(a5)
   move.w   #0,BLTCON1(a5)
   move.w   #64*height+width/2,BLTSIZE(a5)  ; trigger blitter
```

There is *nothing* stopping an interrupt, or copper, triggering a blitter operation between the WaitBlit() call and your final BLTSIZE blitter trigger. This can lead to total system blowup.

Code that may, by luck, work on standard speed machines may die horribly on faster processors due to timing differences causing this type of problem to occurr.

You can prevent this by using OwnBlitter()

The safest way to avoid this is to keep all your blitter calls together, use the copper exclusively, or write a blitter-interrupt routine to do your blits for you, which is very good because you avoid getting stuck in a waitblit-loop.

Always use the graphics.library WaitBlit() routine for your end of blitter code. It does not change any registers, it takes into account any revision of blitter chip and any unusual circumstances, and on an Amiga 1200 will execute faster (because in 32-bit ROM)

than any code that you could write in chipram.

********** 7.3 Blitter - Blitter Speeds **********

BLITTER SPEEDS. (from the Hardware Reference Manual)

Some general notes about blitter speeds. These numbers are for the
blitter only, in 16-bit chip ram.

$$\text{time taken} = \frac{n * H * W}{7.09}$$ (7.15 for NTSC)

time is in microseconds. H=blitheight,W=blitwidth(#words),n=cycles

n=4+....depends on # DMA-channels used

A: +0 (this one is free!)
B: +2
C or D: +0 In line-mode, every pixel takes 8 cycles.
C and D: +2

So, use A,D,A&D for the fastest operation.
Use A&C for 2-source operations (e.g. collision check or so).

********** 7.4 Blitter - Calculating LF Bytes **********

Instead of calculating your LF-bytes all the time you can do this

```
A  EQU   %11110000
B  EQU   %11001100
C  EQU   %10101010
```

So when you need an lf-byte you can just type:

```
   move.w   #(A!B)&C,d0
```

********** 7.5 Blitter - Blitter Clears ***********

Blitter clears

If you use the blitter to clear large areas, you can generally
improve speed on higher processors (68020+) by replacing it by
a cache-loop that clears with movem.l instead:

```
        moveq  #0,d0
        moveq  #0,d1
        moveq  #0,d2
        moveq  #0,d3
        moveq  #0,d4
        moveq  #0,d5
        moveq  #0,d6
        sub.l  a0,a0
        sub.l  a1,a1
        sub.l  a2,a2
        sub.l  a3,a3
        sub.l  a4,a4
        sub.l  a5,a5

        lea    EndofBitplane,a6
        move.w #(bytes in plane/156)-1,d7
.Clear
        movem.l d0-d6/a0-a5,-(a6)
        movem.l d0-d6/a0-a5,-(a6)
        movem.l d0-d6/a0-a5,-(a6)
        dbf d7,.Clear

; final couple of movems may be needed to clear last few bytes of screen...
```

This loop was (on my 1200) almost three times faster than
the blitter.

With 68000-68010 you can gain some time by NOT using blitter-
nasty and the movem-loop.

********** 8. CDTV **********

Programming CDTV/A570

Until now there has been no CDTV documentation available to the public... Well, here are a few tips.....

********** 8.1 CDTV - Using CDTV.Device ***********

Using cdtv.device

The CDTV can be controlled by the cdtv.device, which is
a standard Amiga device.

Open the cdtv.device as standard, and issue commands to it
to play audio, read data, etc...

Examine cdtv.i , included in the source directory.

For example: To play track 2 on an audio CD in a CDTV, use
the following:

```
   include  "cdtv.i"

   ...... your code here ......

   move.l   MyCDTVRequest,a1        ; set this up as for any
                                    ; other device (eg trackdisk.device)

   move.w   #CDTV_PLAYTRACK,IO_COMMAND(a1)

   move.l   #2,IO_OFFSET(a1)        ; track number
   move.l   #1,IO_LENGTH(a1)        ; number of tracks to play

   move.l   4.w,a6
   jsr      _LVOSendIO(a6)          ; send command
```

If you need to gain extra memory, you can shut down the cdtv.device
(apparently) by issuing a CDTV_STOP command to the device.

********** 8.2 CDTV - Checking for A570 **********

Checking for A570

You can tell if you are running on an A570 (as opposed to CDTV)
by checking for "A690ID" with the FindResident function.

```
   FindResident("A690ID")
```

If it returns NULL then it's not A570, if it returns an address
then its an A570

********** 8.1a CDTV - FindResident() ***********

FindResident - find a resident module by name

```
resident = FindResident(name)
   D0          -96       A1

struct Resident *FindResident(STRPTR);
```

Search the system resident tag list for a resident tag ("ROMTag") with the given name. If found return a pointer to the resident tag structure, else return zero.

IN:
name - pointer to name.

OUT:
resident - pointer to the resident tag structure (or NULL)

********** 8.3 CDTV - AmigaCD 32 Information **********

As HTC7 was going to press the AmigaCD32 had been launched in Germany, and UK launch is imminent (July 16th):

AmigaCD32 is:

   68020 14Mhz processor unit, double speed CD-ROM. Will run AmigaCD, CD+G, CDTV and CD Audio discs. It contains AGA chipset and Kickstart 3.1.

   It has two joystick/mouse ports, Composite video, RF (PAL), S-VHX and AUX (A4000 keyboard port). There are *NO* other AMiga ports. No RGB (so no monitors...) no Serial or Parallel (so no Parnet!!!!), and most strange of all - no floppy disk drive port :-(

   It is being sold as a games console to rival Nintendo and Sega.

The most interesting new feature is a new piece of hardware to do *fast* chunky to planar pixel conversion. Hopefully this will be fitted to the Amiga 1200 and 4000 in time...

Programming is done in the same way as any other Amiga model. There are some new libraries and devies, including lowlevel.library, that allows direct control of the new joypad controller (with 10 buttons)

********** 9. Copper **********

Programming the Copper coprocessor

The copper is probably the most wonderful part of the Amiga architecture. It's so simple to program (it has only three different instructions), yet it is powerful enough to create the most complex displays.

Currently the Amiga OS is very limited in allowing programmers access to the copper, and as most demos rely on the power of the copper, most demo coders ignore the OS and build their own displays directly with their own copperlists.

********** 9.1 Copper - Proper Copper Startup ***********

Proper Copper startup

(Please look at the startup example code in startup.asm).

If you are going to use the copper then this is how you should set it up. The current workbench view and copper address are stored, and then the copper enabled. On exit the workbench view is restored.

This guarantees(*) your demo will run on an AGA (Amiga 1200/4000) machine, even if set into some weird screen mode before running your code.

Otherwise under AGA, the hardware registers can be in some strange states before your code runs, beware!

The LoadView(NULL) forces the display to a standard, empty position, flushing the rubbish out of the hardware registers: Note. There is a bug in the V39 OS on Amiga 1200/4000 and the sprite resolution is *not* reset, you will have to do this manually if you use sprites, see Resetting AGA sprite information .

Two WaitTOF() calls are needed after the LoadView to wait for both the long and short frame copperlists of interlaced displays to finish.

It has been suggested to me that instead of using the GfxBase gb_ActiView I should instead use the Intuition ib_ViewLord view. This will work just as well, but there has been debate as to whether in the future with retargetable graphics (RTG) this will work in the same way. As the GfxBase is at a lower level than Intuition, I prefer to access it this way (but thank's for the suggestion Boerge anyway!). Using gb_ActiView code should run from non-Workbench environments (for example, being called from within Amos) too...

* - Nothing is ever guaranteed where Commodore are involved. They may move the hardware registers into chipram next week :-)

********** 9.1a Copper - LoadView() ***********

LoadView -- Use a (possibly freshly created) copper
            list to create the current display.

LoadView(View)
 -222   A1

void LoadView(struct View *);

IN:
   View - a pointer to the View structure which contains the
   pointer to the constructed coprocessor instructions list, or NULL.

If the View pointer is non-NULL, the new View is displayed,
according to your instructions. The vertical blank routine
will pick this pointer up and direct the copper to start
displaying this View.

If the View pointer is NULL, no View is displayed, and the hardware
defaults back to standard chipset defaults (mostly).

Even though a LoadView(NULL) is performed, display DMA will still be
active. Sprites will continue to be displayed after a LoadView(NULL)
unless an OFF_SPRITE is subsequently performed.

********** 9.1b Copper - WaitTOF() **********

WaitTOF -- Wait for the top of the next video frame.

```
WaitTOF()
  -270
```

```
void WaitTOF( void );
```

Wait for vertical blank to occur and all vertical blank
interrupt routines to complete before returning to caller.

********** 9.2 Copper - Copper Wait Bugs **********

Copper Wait Commands

The Hardware Reference manual states a copper wait for the start of line xx is done with:

$xx01,$fffe

However (as many of you have found out), this actually triggers just before the end of the previous line (around 4 or 5 low-res pixels in from the maximum overscan border).

For most operations this is not a problem (and indeed gives a little extra time to initialise stuff for the next line), but if you are changing the background colour ($dff180), then there is a noticable 'step' at the end of the scanline.

The correct way to do a copper wait to avoid this problem is

$xx07,$fffe.

This just misses the previous scanline, so the background colour is changed exactly at the start of the scanline, not before.

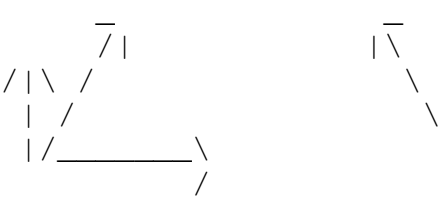
********** 10. Vectors **********

How to code vectordemos

********** 10.0 Vectors - Introduction ***********

An introduction by Asterix of Movement ...
==

 This text is an addition to How to code written by Comrade J of SAE.
It was written by Carl-Henrik Skårstedt during his easter holidays.
Hi ho to all friends of movements....
Any comments on this text/additions to vectors.txt should be Emailed to
mnlcst@cs.umu.se, or by mail: Rullstensg6-210,90655Ume},Swe
If you think I should be clearer on some points, or if you think I've
totally forgotten something, just report this for later issues..



********** 10.1 Vectors - Preface **********

1. Preface

The sources of this text has more or less indirectly been some books from my school. Some sources worth mentioning are:
Elementary Linear Algebra (by Howard Anton, released by John Wiley)
Calculus - A complete course (By Robert K. Adams)
The DiscWorld series (by T. Pratchett)

By reading this text, you should also be able to know what it is you're doing. Not just converting given formulas to 680x0 code, but also know what the background of it is. If you know the theory behind your routine, you also know how to optimize it! NO text will here mention GLENZ-vectors, since they are amazingly depressive.

This text is meant for democoders on all computers that supports a good graphic interface, which is fast enough to do normal concave objects in one frame (not PC).
sqr() means sqare root in this text.

I'm curious about what support Commodore has for this kind of programming in their latest OS, it could be a great Idea if rotations etc that used many multiplications was programmed in ROM. The methods described are used by most well-known demo-coders in "vector" demos.

I've coded Blue House I+2, most of Rebels Megademo II, my own fantasic and wonderful cruncher (not released), Amed (also not released), some intros, and the rubiks snake in Rebels Aars-intro, and the real slideshow from ECES. Sorry for most of my demos not working on other machines than real A500's, but that's the only computer I've used for bugtesting.

The meaning of this text is that it shall be a part of How To Code.txt and that the same rules works for this text as for that.

Sourcecodes should work with most assemblers except for Insert sorting, which needs a 68020 assembler.

Hi to all my friends who really supported me by comments like:
"How can you write all that text?"
"Who will read all that?"
"Can I have it first, so I can get more bbs-access?"
"Why are you writing that?"
"I want to play Zool!" (-My youngest brother)
"My dog is sick..."
"You definitely have the right approach to this!"
"" (-Commodore Sweden)
(But in swedish of course!)

The reason why Terry Pratchetts DiscWorld series is taken as a serious source is that he is a great visualizer of strange mathematical difficulties. If you ever have

problems with inspiration, sit back, read and try to imagine
how demos would look like in the DiscWorld... (Glenz-Turtles?)

Now read this text and impress me with a great AGA-demo...

"Death to the pis" /T. Domar

********** 10.2 Vectors - Introduction to vectors **********

2. Introduction to vectors

What is a vector?

If you have seen demos, you have probably seen effects that is called, in a loose term, vectors. They may be balls, filled polygons, lines, objects and many other things.
The thing that is in common of these demos are the vector calculations of the positions of the objects. It can be in one, two or three Dimensions (or more, but then you can't see the ones above 3)

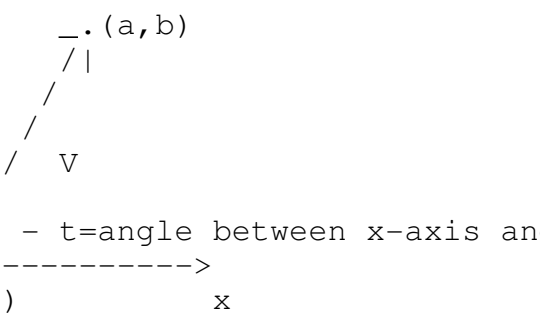
You can for example have a cube. Each corner on the cube represent a vector TO the center of rotation.
All vectors go FROM something TO something, normally we use vectors that goes from a point (0,0) to a point (a,b).
This vector has the quantity of (a,b).

Definition of vector:
A QUANTITY of both VALUE and DIRECTION.

or, in a laymans loose terms: a line.
A line have a length that we can call r, and a direction we can call t.
We can write this vector (r,t) = (length,angle).
But there is also another way, which is more used when dealing with vector objects with given coordinates.

The line from (0,0) to (x,y) has the length sqr(x*x+y*y) and that is the VALUE of the vector. The direction can be seen as the angle between the x-axis and the line described by the vector.

If we study this in two dimensions, we can have an example vector as following:



We can call this vector V, and, as we can see, it goes from the point (0,0) and (a,b). We can denote this vector as V=(a,b).
Now we have both a value of V (The length between (0,0) and (a,b)) and a direction of it (the angle in the diagram)

If we look at the diagram, we can see that the length of the vector can be computed with pythagoras theorem, like:
r=sqr(a*a+b*b)

and t is the angle (Can be calculated with t=tan(y/x))

Three Dimensions?

Now, if we have seen what a vector is in two dimensions, what is a vector in three?

In three dimensions, every point has three coordinates, and so must then the vector have.

V=(a,b,c)

Now the length of the vector becomes:
r=sqr(a*a+b*b+c*c)

What happens to the angle now?

Here we can have different definitions, but let's think a little. If we start with giving ONE angle, we can only reach points on one PLANE, but we want to get a direction in SPACE.

If we try with TWO angles, we will get a better result. One angle can represent the angle between the z-axis and the vector, the other the rotation AROUND the z-axis.

For more problems in this area (there's many) study calculus of several variables and specially polar transformations in triple integrals, or just surface integrals in vector fields.

2.1 Vector operations:

(if you have two, or one dimension you have two or one variable instead of three. if you have more you have ofcourse as many variables as dimensions)

* The SUM of two vectors (U=V+W) are defined as:
  V=(vx,vy,vz), W=(wx,wy,wz)=>
  => U=(vx+wx,vy+wy,vz+wz)

* The negation of a vector U=-V is defined as
  V=(x,y,z) => U=(-x,-y,-z)

* The differance between two vectors U=V-W are defined as
  U=V+(-W)

* A vector between two points (FROM P1(x1,y1,z1) TO P2(x2,y2,z2)) can be computed:
  V=(x2-x1,y2-y1,z2-z1,...)
  (V goes FROM P1 TO P2)

* A vector can be multiplied by a constant like:
  U=k*V
  (x*k,y*k,z*k)=k*(x,y,z)

* A coordinate system can be "Translated" to a new point with the translation formula:
  x'=x-k
  y'=y-l

z'=z-m
Where (k,l,m) is the OLD point where the NEW coordinate system
should have its point (0,0,0)
This is a good operation if you want to ROTATE around A NEW POINT!

* A vector can be rotated (Check chapter 4)
The vector is always rotated around the point (0,0,0) so you
may have to TRANSLATE it.

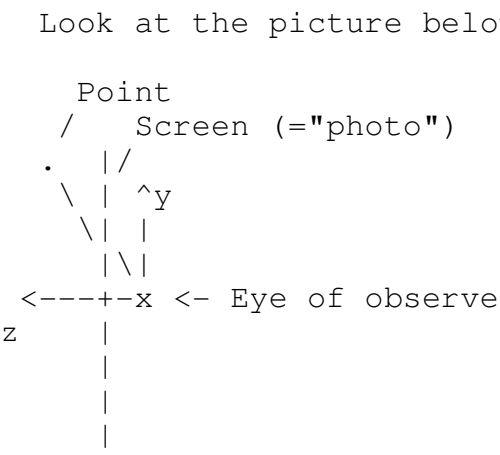
* We can take scalar product and cross-product of vectors
(see any book about introduction to linear algebra
for these. everything is evaluated in this text, so you
don't have to know what this is)

3. Coding techniques

Presenting a three dimensional point on a two dimensinal screen

Assume that you have a point in space (3d) that you want to take a photo of. A photo is 2d, so this should give us some sort of an answer.

Look at the picture below:



Inspecting this gives us the following formula:

Projected Y = Distance of screen * Old Y / (Distance of point)
(The distances is of course the Z coordinates from the Eyes position)

And a similar way of thinking gives us the projection of X.

```
New Y=k*y/(z+dist)
    X=k*x/(z+dist)
```

(where k is a constant for this screen, dist is the distance from the ROTATION point to the EYE on the Z-axis)

A way of presenting real numbers with Integers

Until now we have only seen a lot of formulas, but how can we use them in Assembler where we only can have bytes/words/longwords? (If you don't have a FPU, and only want people with FPU's to be able to see your demos)

For 68000 coding (compatible with all better processors) it is comfortable to be able to do multiplictations etc. with words (680x0,{x>=2} can do it with longwords, but this won't work very good with lower x's).

But we need the fract parts of the numbers too, so how do we do? We can try to use numbers that are multiplied by a constant p. Then we can do the following operation:

[cos(a)*p] * 75   (for example from a list with cos(x) mult. with p)

But as you can see this number grows for each time we do another
multiplication, so what we have to do is to divide by p again:

[cos(a)*p] * 75 / p

If you are a knower of digital electronics, you say "Oh no,
not a division that takes so long time". But if you choose
p carefully (i.e. p = 2 or 4 or 8 ...) you can use
shifting instead of clean division. Look at this example:

```
      mulu    10(a0),d0       ;10(a0) is from a list of cos*256 values
      asr.l   #8,d0           ;and we "divide" by 256!
```

Now we have done a multiplication of a fixed point number!
 (A hint to get the error a little smaller:
  clear a Dx-register and use an addx after the asr,
  and you will get a round-off error instead:

```
       moveq  #0,d7
       :
       :
       mulu   10(a0),d0
       asr.l  #8,d0
       addx.l d7,d0
       :
       rts
```
  This halves the error!)

 The same thinking comes in with divisions, but in the other way:

```
      :
      ext.l   d0
      ext.l   d1
      asl.l   #8,d0           ;"Multiply" by 256
      divs    d1,d0           ;and divide by z*256 ...
      :
      rts
```

Additions and subtractions are the same as normal integer
operations: (no shifting needed)

```
      :
      add.w   10(a0),d0
      :

      :
      sub.w   16(a1),d1
      :
```

So, With multiplications you MUL first, then LSR.
     With divisions you LSL first, then DIV.

If you wish to have higher accuracy with the multiplications,
the 68020 and higher processors offers a cheap way to do
floating point operations (32-bit total) instead.
You can also do integer multiplications 32*32->32,
and use 16-bit coses and sins instead, which enables
you to use 'swap' instead of 'lsr'.

How can I use Sin and Cos in my assembler code?

The easiest and fastest way is to include a sinus-list in you program. Make a basic-program that counts from 0 to 2*pi, for example 1024 times. Save the values and include them into your code.

If you have words and 1024 different sinus values then you can get sinus and cosinus this way:

```
      lea     sinuslist(pc),a0        ;sinuslist is calculated list
      and.w   #$7fe,d0                ;d0 is angle
      move.w  (a0,d0.w),d1            ;d1=sin(d0)
      add.w   #$200,d0
      and.w   #$7fe,d0
      move.w  (a0,d0.w),d0            ;d0=cos(original d0)
      :
      :
```

Your program could look like this: (AmigaBasic, HiSoft basic)
(NEVER use AmigaBasic on other processors than 68000)

```
 open "ram:sinuslist.s" for output as 1
 pi=3.141592654#
 vals=1024
 nu=0
 pretxt=chr$(10)+chr$(9)+chr$(9)+'dc.w'+chr$(9)
  for L=0 to vals
   angle=L/vals*2*pi
   y='$'+hex$(int(sin(angle)*255.4))
   nu=nu+1
   if nu=8 then print #1,pretxt;:nu=0 else print #1,',';
   print #1,y$;
  next L
 close 1
```

You can of course do a program that calculates the sins in assembler code, by using ieee-libs or coding your own floating point routines.
the relevant algoritm is... (for sinus)

```
indata: v=angle (given in radians)
        Laps=number of terms (less=faster and more error, integer)

  1> Mlop=1
     DFac=1
     Ang2=angle*angle
     Talj=angle
     sign=1
     Result=0
  2> FOR terms=1 TO Laps
  2.1> Temp=Talj/Dfac
  2.2> Result=sign*(Result+Temp)
  2.3> Talj=Talj*Ang2
  2.4> Mlop=Mlop+1
  2.5> Dfac=Dfac*Mlop
  2.6> sign=-sign
  3> RETURN sin()=Result
```

where the returned sin() is between -1 and 1...
The algorithm uses MacLaurin polynoms, and are therefore
recommended only for values that are not very far away from 0.

4. The rotation of vectors

* In two dimensions

Now we know what a vector is, and we want to rotate it.
This is very simple, if we have a vector given
with lenght and angle, we just add the rotation-angle to the
angle and let the length be like it is:
rotate V=(r,t) with a -> V'=(r,t+a)

But normally we don't have this simple case, we have a
vector given by two coordinates like:
V=(x,y) where x and y are coordinates in the xy-plane.

In THIS text we denote the rotation of a vector V=(r,t) with
rot(V,a). With this I mean the rotation of the vector V with the
angle a.

The rotation of this vector could have been done
by transforming V to a vector of a length and a direction,
but since this involves taking squares, tangens, squareroots
etc., we would like a faster method.
Here is where trigonometry comes in.

But let us first assume that we have a vector V=(x,0)
what would be the rotation of this vector?

```
     V
----------->
```

Now, let us rotate with an angle of a:

```
    _
/|\y' /|
 |   /
 |V'/
 | /
 |/\a x'
  ----->
```

What is the new vectors composants (x',y') ?

Remember these "definitions":

Cosine:
 Hypotenuse/side close to angle

Sine:
 Hypotenuse/side not close to angle

```
       ,
      /|
Length>/ |< Length * sin(a)
     /a |
    '---+
```

Length * cos(a)

If we put in this in the original rotation formula (V'=rot(V,a)=V(r,t+a))

we can see that we can convert r and t to x and y with:

```
  x=r*cos(t)
  y=r*sin(t)
```

Let's get back to our problem of the rotated vector V=(x,0).
Here is r=x (=sqrt(x*x+0*0)), t=0 (=arctan(0/x)
if we put this in our formula we get:

V=(r,t) if r=x, t=0

If we rotate this vector with the angle a we get:

V=(r,t+a)

And if we translate back to our coordinate denotion:

```
V=(r*cos(t+a),r*sin(t+a))=(x*cos(a),x*sin(a))
                         ^We insert x=r, t=0
```

And that is the formula for rotation of a vector that has no y-composant.
For a vector V=(0,y) we get:
r=y, t=pi/2 (=90 degrees) since we now are in the y-axis,
which is 90 degrees from the x-axis.

```
V=(r,t) => V'=(r,t+a) => V'=(r*cos(t+a),r*sin(t+a)) =>
V'=(y*cos(pi/2+a),y*sin(pi/2+a))
```

Now, there are a few trigonometric formulas that says that cos(pi/2+a)=
=sin(a) and sin(pi/2+a)=-cos(a)

We get:
V'=(y * sin(a) , y * (-cos(a)))

But if we look in the general case, we have a vector V that
has both x and y composants. Now we can use the single-cases
rotation formulas for calculating the general case with
an addition:

```
  Vx'=rot((x,0),a) = (x*cos(a)         ,x*sin(a))
+ Vy'=rot((0,y),a) = (        +y*sin(a),        -y*cos(a))
-------------------------------------------------------
  V' =rot((x,y),a) = (x*cos(a)+y*sin(a),x*sin(a)-y*cos(a))
```

(Vx' means rotation of V=(x,0) and Vy' is rotation of V=(0,y))
And we have the rotation of a vector given in coordinates!

FINAL FORMULA OF ROTATION IN TWO DIMENSIONS

```
.. .
 . rot( (x,y), a)=( x*cos(a)+y*sin(a) , x*sin(a)-y*cos(a) )
       x-composant ^^^^^^^^^^^^^^^^    ^^^^^^^^^^^^^^^^^ y-composant
```

* Three dimensions

Now we are getting somewhere!

In the 2 dimensions case, we rotated x and y coordinates,
and we didn't see any z coordinates that changed.
Therefore we call this a rotation around the Z axis.

Now, the simpliest thing to do in three dimensions is to
still do the same thing, just rotate around any axis
to get the new coordinate. Leave out the variable that
represents the coordinate of the current rotation-axis,
and you can use the very same expression.

If you want to rotate only one or two coordinates, you can use
the normal method of rotation, because then you
won't have to calculate a 3x3 transformation matrix.
But if you have more points, I recommend the
optimized version.

But there are optimizations in this field, but let's first
look at ONE way to do this:

NORMAL METHOD OF ROTATING A VECTOR WITH THREE GIVEN ANGLES IN 3D:

Assume we want to rotate V=(x,y,z) around the z-axis
with the angle a, around y with b and around x with c.
The first rotation we do is around the Z-axis:
U=(x,y) (x,y from V-vector) =>
=> U'=rot(U,a)=rot((x,y),a)=(x',y')

Then we want to rotate around the Y-axis:
W=(x',z) (x' is from U' and z is from V) =>
=> W'=rot(W,b)=rot((x',z),b)=(x'',z')

And finally around the X-axis:
T=(y',z') (y' is from U' and z' is from W') =>
=> T'=rot(T,c)=rot((y',z'),c)=(y'',z'')

The rotated vector V' is the coordinate vector
(x'',y'',z'') !

With this method we can extend out rot-command to:

V''= rot(V,angle1,angle2,angle3) where V is the original vector!
(V''= rot((x,y,z),angle1,angle2,angle3))

I hope that didn't look too complicated.
As I said, there are optimizations of this method.
These optimizations can be skipping one rotation of the
above ones, or some precalculations.

ORDER is very important. You won't get the same answer
if you rotate X,Y,Z with the same angles as before.

Optimizations:
==============
For xyz vectors we can write the equations to
form the rotations:

let c1=cos(angle1), c2=cos(angle2), c3=cos(angle3),
s1=sin(angle1), s2=sin(angle2), s3=sin(angle3)

(x*cos(a)+y*sin(a),x*sin(a)-y*cos(a))

```
 x' = x*c1+y*s1
 y' = x*s1-y*c1

 x''= x'*c2+z*s2        <- Rotated x-coordinate
 z' = x'*s2-z*c2

 y''= y'*c3+z'*s3     <- Rotated y-coordinate
 z''= y'*s3-z'*c3        <- Rotated z-coordinate
```

which gives:

```
 x''= (x*c1+y*s1)*c2+z*s2= c2*c1 *x + c2*s1 *y + s2 *z
      ^^^^^^^^^^^=x'       ^^^^^ xx   ^^^^^ xy   ^^ xz

 y''= (x*s1-y*c1)*c3+((x*c1+y*s1)*s2-z*c2)*s3=
      c3*s1 *x - c3*c1 *y + s3*s2*c1 *x + s3*s2*s1 *y - s3*c2 *z=

      (s3*s2*c1+c3*s1) *x + (s3*s2*s1-c3*c1) *y + (-s3*c2) *z
      ^^^^^^^^^^^^^^^^ yx   ^^^^^^^^^^^^^^^^ yy   ^^^^^^^^ yz

 z''= (x*s1-y*c1)*s3-((x*c1+y*s1)*s2-z*c2)*c3=
      s3*s1 *x - s3*c1 *y - c3*s2*c1 *x - c3*s2*s1 *y + c3*c2 *z=

      (-c3*s2*c1+s3*s1) *x + (-c3*s2*s1-c3*c1) *y + (c3*c2) *z
      ^^^^^^^^^^^^^^^^^ zx   ^^^^^^^^^^^^^^^^^ zy   ^^^^^^^ zz
```

Now, look at the pattern of the solutions,
for x'' we have calculated something times the original (x,y,z),
the same for y'' and z'', What is the connection?

Say that you rotate many given vectors with three angles that are the
same for all vectors, then you get this scheme of multiplications.
When you rotated as above you had to use twelve multiplications
to do one rotation, but now we precalculate these 'constants'
and manage to get down to nine multiplications!
                          ^^^^

FINAL FORMULA FOR ROTATIONS IN THREE DIMENSION WITH THREE ANGLES

(x,y,z is the original (x,y,z) coordinate.
 c1=cos(angle1), s1=sin(angle1), c2=cos(angle2) and so on...)

If you want to rotate a lot of coordinates with the same angles you
first calculate these values:

```
xx=c2*c1
xy=c2*s1
xz=s2
yx=c3*s1+s3*s2*c1
yy=-c3*c1+s3*s2*s1
yz=-s3*c2
zx=s3*s1-c3*s2*c1;s2*c1+c3*s1
zy=-s3*c1-c3*s2*s1;c3*c1-s2*s1
zz=c3*c2
```

Then, for each coordinate, you use the following multiplication
to get the rotated coordinates:

```
x''=xx * x + xy * y + xz * z
y''=yx * x + yy * y + yz * z
z''=zx * x + zy * y + zz * z
```

So, you only have to calculate the constants once for every new angle,
and THEN you use nine multiplications for every point you wish to
rotate to get the new set of points.

Look in the end of this text for an example of how this can be
implemented in 68000-assembler.

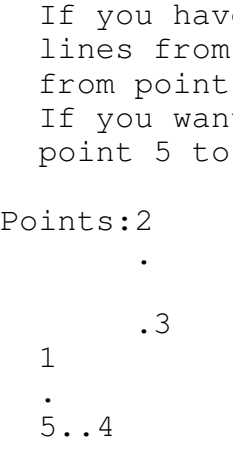
If you wish to skip on angle, you can optimize further.
if you want to remove angle3, set c3=1 and all s3=0
and put into your constant-calculation and it will be
optimized for you.

What method you want to use depends of course on how much you want to
code, but I prefer the optimized version since it's more
to be proud of... If you only rotate a few points with the same
angles, the first (non-optimized) version might be the choice.

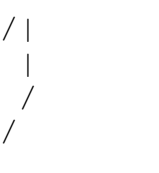
If you want to, you can check that the transformation matrix has
a determinant equal to 1.

5. Polygons!

The word "polygon" means many corners, which means that it has a lot of points (corners) with lines drawn to. If you have, for example, 5 points, you can draw lines from point 1 to point 2, from point 2 to point 3, from point 3 to point 4 and from point 4 to point 5. If you want a CLOSED polygon you also draw a line from point 5 to point 1.



Open polygon of points above:



Closed polygon of points above:

"Filled vectors" is created by drawing polygons, and filling inside. Normally the following algorithm is used:

First you define all "corners" on the polygon as vectors, which allows you to rotate it and draw it in new angles, and then you draw one line from point 1 to point 2, and so on. The last line is from point 5 to point 1. When you're finished you use a BLITTER-FILL operation to fill the area.

You will need a special line drawing routine for drawing these lines so the BLITTER-FILL works, I have an example of a working line-drawing routine in the appendices (K-seka! Just for CJ!). Further theory about what demands there are on the line drawing routine will be discussed later (App. B 2).

There are also other ways to get a filled area (mostly for computers without blitter, or for special purposes on those that have) Information about that will be in later issues.

Creating objects from polygons

An object is in this text a three-dimensional thing created with polygons. We don't have to think about what's inside, we just surround a mathematically defined sub-room with polygons.

But what happends to surfaces that is on the other side of the object? and if there are hidden "parts" of the object, what can we do about them?

We begin with a cube, it is easy to imagine, and also the rotation of it. we can see that no part of the cube is over another part of the cube in the viewers eyes. (compare, for example, with a torus, where there are sometimes parts that hides other parts of the same object) Some areas are of course AIMING AWAY FROM THE VIEWER, but we can calculate in what direction the polygon is facing (to or from the viewer)

Always define the polygons in objects in the same direction (clockwise or non-clockwise) in all of the object. imagine that you stand on the OUTSIDE MIDDLE of the plane, and pick all points in a clockwise order. Which one you start with has nothing to do with it, just the order of them.

Pick three points from a plane (point1, point2 and point 3) If all three points are not equal to any of the other points, these points define a plane.
You will then only need three points to define the direction of the plane. Examine the following calculation:

```
c=(x3-x1)*(y2-y1)-(x2-x1)*(y3-y1)
```

(This is AFTER 3d->2d projection, so there's no z-coordinate. If you want to know what this does, look in appendix b)

This needs three points, which is the minimum number of coordinates a polygon must be, to not be a line or a point (THINK). This involves two multiplications per plane, but that isn't very much compared to rotation and 3d->2d projection.

But let us study what this equation gives:
If c is negative, the normal vector of the plane which the three points span is heading INTO the viewer (= The plane is fronting the viewer => plane should be drawed)...
If c is POSITIVE, the normal vector of the plane is heading AWAY from the viewer (= The plane cannot be seen by the viewer => DON'T draw the plane) ...

But to question 2, what happends if parts of the object covers OTHER parts of the object...

Convex and concave objects

"Definitions"

A convex object has NO parts that covers other parts of the same object, viewed from all angles.

A concave object has parts that covers other parts of the same object, viewed from some angle.

--

For convex objects, this means that you can draw a straigt
line from every point inside the object to every other point
in the object without having no line that passes outside
the domain of the object.

If you have a CONVEX object, you can draw ALL lines around
the visible planes and then fill with the blitter, because
no drawn polygon will ever cover another polygon.
With some struggle you can also find ways to omit some lines,
since they will be drawn twice.

CONCAVE objects offers some further problems,
the easiest way to use CONCAVE objects is
to split them into smaller CONVEX objects. This works for
all objects, even if you can have some problem doing it.

Of course, you can skip a lot of planes that will be "inside"
the concave object.

When you have splitted the object you simply draw
each convex object into a TEMPORARY memory area,
and treat it like a VECTORBALL (Sorting and stuff),
Which should be discussed in later parts of this text.

The Z coordinate can be taken from the middle of all z-values
in the object (The sum of all Z's in the object divided by
the number of coordinates)

When you're sorting the objects, you can sometimes have problems
with parts of the concave object getting in the wrong order
since you've picked a point at random from the OUTSIDE of the
convex object, which the current object is sharing with
another convex object. One way to solve this problem
is to take a middle point that is inside the convex object,
by adding all the Z-values around the object and dividing
by the number of coordinates that you have added.
In this case, you should take points from at least two planes in
the object.

Object optimization

Assume that you have a CONVEX object.
If it is closed, you have almost as few points as you have
planes. If you have a list to every coordinate
that exist (no points are the same in this list) that
for each polygon shows what point you should fetch for
this coordinate, you can cut widely on the number of
ROTATIONS.
For example:

```
/* A cube */
/* order is important! Here is clockwise */

 end_of_plane=0

 pointlist
```

```
    dc.l pt4,pt3,pt2,pt1,end_of_plane
    dc.l pt5,pt6,pt2,pt1,end_of_plane
    dc.l pt6,pt7,pt3,pt2,end_of_plane
    dc.l pt7,pt8,pt4,pt3,end_of_plane
    dc.l pt8,pt5,pt1,pt4,end_of_plane
    dc.l pt5,pt6,pt7,pt8,end_of_plane
pt1 dc.w -1,-1,-1
pt2 dc.w 1,-1,-1
pt3 dc.w 1,-1,1
pt4 dc.w -1,-1,1
pt5 dc.w -1,1,-1
pt6 dc.w 1,1,-1
pt7 dc.w 1,1,1
pt8 dc.w -1,1,1
```

Now, you only have to rotate the points pt1-pt8, which is eight points. If you had computed four points for each plane, you would have to compute 24 rotations instead.

6. Planes in three dimensions

Lightsourcing

Lightsourcing is a way to find out how much light a plane recieves from either a point of light (spherical) or a plane of light (planar). If the colour of the plane represents the light that falls on it, the object will be a bit more realistic.

What we are interested in is the ANGLE of the VECTOR from the NORMAL of the plane to the LIGHTSOURCE (=point of light) (this is for a spherical lightsource, like a lamp or something. If you are interested in planar light, like from the sun, you are interested in the ANGLE between the NORMAL of the plane and the LIGHTSOURCE VECTOR)

We are interested of the COSINE of the given angle.

Anyway, to get the normal of the plane you can pick three points in the polygon, create two vectors of these.

Example:

* we pick (x1,y1,z1) and (x2,y2,z2) and (x3,y3,z3)
we create two vectors V1 and V2:
V1=(x2-x1,y2-y1,z2-z1)
V2=(x3-x1,y3-y1,z3-z1)

To get the normal of these we take the cross product of them:

$$N = V1xV2 = \begin{vmatrix} i & j & k \\ x2-x1 & y2-y1 & z2-z1 \\ x3-x1 & y3-y1 & z3-z1 \end{vmatrix} =$$

$$\begin{matrix} & n1 & & & n2 & \\ * & = ((y2-y1)*(z3-z1)-(y3-y1)*(z2-z1), & -((x2-x1)*(z3-z1)-(x3-x1)*(z2-z1)), & & & \\ * & ,(x2-x1)*(y3-y1)-(x3-x1)*(y2-y1)) & & & & \\ & n3 & & & & \end{matrix}$$

Now, we have N. We also have the LIGHTSOURCE coordinates (given)

To get COS of the ANGLE between two vectors we can use the scalar product between N and L (=lightsource vector) divided by the length of N and L:

$\langle N,L \rangle / (||N|| * ||L||) =$

* (n1*l1+n2*l2+n3*l3)/(sqr(n1*n1+n2*n2+n3*n3)*sqr(l1*l1+l2*l2+l3*l3))
|
* (can be (n1*l1+n2*l2+n3*l3)/k if k is a precalculated constant)

This number is between -1 and 1 and is cos of the angle between the vectors L and N. the SQUARE ROOTS take much time, but if you keep the object intact (do only rotations/translatins etc.) and always pick the same points in the object, then ||N|| is intact and can be precalculated.

If you make sure the length of L is always 1, you won't have
to devide by this, which saves many cycles.

The number will, as said, be between -1 and 1. You may have
to multiply the number with something before dividing
so that you have a larger range to pick colours from.
If the number is negative, set it to zero.

The number can be NEGATIVE when it should be POSITIVE,
this is because you took the points in the wrong order,
but you only have to negate the result instead.

If you didn't understand a thing of this, look on the formulas
with a '*' in the border. n1 means the x-coordinate of N, n2
the y-coordinate and so on, and the same thing with L.

********** 10.7 Vectors - Sorting Algorithms **********

Special techniques - Sorting Algorithms

When you come to sorting, most books begin with "Bubble-sorting"
Bubble sorting is enourmously slow, and is described here only
for explaining terms. But I don't advise you to code routines
that uses this method since it's SLOOOOOOOOOOOW....
A better way is to use Insert Sorting (which, in fact, is sorting,
Acro!) or Quick Sorting or whatever you can find in
old books (you have them, I'm sure!) about basic/pascal/c/
or whatever (and even a few assembler books!!!) contains
different methods of sorting. Just make sure you don't use
BUBBLE SORTING!!!!!

Method 1) Bubble sorting

Assume that you have a list of VALUES and WEIGHTS.
The heaviest weights must fall to the bottom, and bringing the
VALUES with it. The values in this case can be the
2d->3d projected x and y coordinates plus bob information.
The Weights can be the Z coordinates before projection.

Begin with the first two elements, check what element
is the HEAVIEST, and if it is ABOVE the lighter element,
move all information connected with the WEIGHT and the
WEIGHT to the place where the light element was,
and put the light data where the heavy was.
(This operation is called a 'swap' operation)

Step down ONE element and check element 2 and 3..
step further until you're at the bottom of
the list.

The first round won't fix the sorting, you will have to
go round the list THE SAME NUMBER OF TIMES AS YOU HAVE
OBJECTS minus one!!!!

The many comparisions vote for a faster technique...

Algorithm:

```
1> FOR outer loop=1 TO Items-1
1.1> FOR inner loop=1 TO Items-1
1.1.1> IF Item(inner loop)>Item(inner loop+1)
1.1.1.1> Swap Item(inner loop),Item(inner loop+1)
```

(Items is the number of entries to sort, Item() is the weight of
the current item)

Method 2) Insert sorting

Assume the same VALUES and WEIGHTS as above.
To do this, you have to select a wordlength for the

sorting-table (checklist) and a size of the checklist.

The wordlength depends on the number of entries you have, and the size of every entry. Normally, it's convienient to use WORDS. The size of the checklist is the range of Z-values to sort, or transformed Z-values. If you, for example, know that your Z-values lies within 512-1023 you can first decrease every z-value by 512, and then lsr' it once, which will give you a checklist size of 256 words. You will also need a second buffer to put your sorted data in, this 2ndBUF will be like a copy of the original list but with the entries sorted.

For this method I only present an algorithm, it's easier to see how it works from that than from some strange text.

checklist(x) is the x'th word in the checklist.

Algorithm:

```
1>  CLEAR the checklist (set all words=0)
2>  TRANSFORM all weights if necessary.
3>  FOR L=0 TO number of objects
3.1> ADD ENTRYSIZE TO checklist(transformed weight)
4>  FOR L=0 TO checklist size-1
4.1> ADD checklist(L),checklist(L+1)
5>  FOR L=0 TO number of objects
5.1> PUT ENTRY at 2ndBUF(checklist(transformed weight))
5.2> ADD ENTRYSIZE TO checklist(transformed weigth)
```

Now, your data is nicely sorted in the list 2ndBUF, the original list is left as it was (except for Z-transformation). (ENTRYSIZE is the size of the entry, so if you have x,y,z coordinates in words, your size is 3 words=6 bytes.) Also try to think a little about what you get when you transform. The subtraction is useful since it minimizes the loops, but lsr-ing the weights take time and makes the result worse. Of course you don't have to scan the list every time, just make sure that you know what the lowest possible and the higest possible weight is.

Method 3) the Quick-Sort

This is another kind of sorting, and here it is most efficient to use pointers, so that each entry have a pointer to NEXT entry.

you can one entry like this:

NEXT OFFSET=word
x,y,z=coordinates.

(offsets are from sortlist start address...)

To access this routine you will have to give a FIRST entry and number of entries. In the original execution, first entry is of course 0 (=first entry) and the number of entries is of course the total number of entries.
You must set all previous/next pointers to link a chain.

Quicksort is recursive, which means that you will have to call the routine from within itself. This is not at all complicated, you just have to put some of your old variables on the stack for safe-keeping.

What it does is this:

+> The first entry in the list is the PIVOT ENTRY.
| For each other ENTRY, we put it either BEFORE or AFTER
| the PIVOT. If it is lighter than the PIVOT we put it BEFORE,
| otherwise we put it AFTER.
| Now we have two new lists, All entries BEFORE the PIVOT,
| and all entries AFTER the PIVOT (but not the pivot itself,
| which is already sorted).
| Now we quicksort All entries BEFORE the pivot separately
+< and then we quicksort all entries AFTER the pivot.
   (We do this by calling on the routine we're already in)
   This may cause problems with the stack if there's too
   many things to sort.

   The recursion loop is broken when there's <=1 entry
   to sort.

   Contrary to some peoples belief, you don't need any extra
   lists to solve this.

Algorithm:

```
Inparameters: (PivotEntry=first element of list
               List size=size of current list)
1> If list size <= 1 then exit
2> PivotWeight=Weight(PivotEntry)
3> for l=2nd Entry to list size-1
3.1> if weight(l) > PivotWeight
3.1.1> insert entry in list 1
3.2> ELSE
3.2.1> insert entry in list 2
4> Sort list 1 (bsr quicksort(first entry list 1, size of list 1))
5> Sort list 1 (bsr quicksort(first entry list 2, size of list 2))
6> Link list 1 -> PivotEntry -> list 2
```

(PivotEntry = FirstEntry, it don't have to be like this, but I prefer it since I find it easier.)

********** 10.8 Vectors - Vector Balls ***********

Special techniques - Vector Balls

Vector balls are simple. Just calculate where the balls are (with rotations, translations or whatever it can be). Sometimes you also calculate the size of the ball and so on.

You don't have to have balls. You can have the Convex parts of an concave filled object, or you can have images of whatever you like. In three dimensions you will have the problem with images (balls or whatever) that should be in front of others because it is further away from you. Here is where SORTING comes in. If you BEGIN blitting the image that is most distant to you, and step closer for each object, you get a 3d-looking screen. The closest image will be the closest.

Normally, you start with clearing the screen you're not displaying at the moment (Parts of it anyway. A person in Silents only cleared every second line...)

Then (while the blitter is working) you start rotating, sorting and preparing to finally bob the images out

and when you've checked that the blitter is finished, you start bobbing out all images, and when the frame is displayed, you swap screens so you display your finished screen the next frame.

Appendix A: Example sources.

1 Optimised rotation matrix calcuation
2 A line draw routine for filled vectors
3 Quicksort in 68000 assembler
4 Insert Sort in 68020 assembler

********** 10.A1 Vectors - Source - Optimized rotation **********

A 1. An example of an optimized rotation matrix calculation
==========

```
* For this routine, you must have a sinus table of 1024 values,
* and three words with angles and a place (9 words) to store
* the transformation matrix.
*     __    .
*  /( |( )|\/ '(|)
* /  )|(|\|/\   |)

Calculate_Constants

                lea     Coses_Sines(pc),a0
                lea     Angles(pc),a2
                lea     Sintab(pc),a1

                move.w  (a2),d0
                and.w   #$7fe,d0
                move.w  (a1,d0.w),(a0)
                add.w   #$200,d0
                and.w   #$7fe,d0
                move.w  (a1,d0.w),2(a0)
                move.w  2(a2),d0
                and.w   #$7fe,d0
                move.w  (a1,d0.w),4(a0)
                add.w   #$200,d0
                and.w   #$7fe,d0
                move.w  (a1,d0.w),6(a0)
                move.w  4(a2),d0
                and.w   #$7fe,d0
                move.w  (a1,d0.w),8(a0)
                add.w   #$200,d0
                and.w   #$7fe,d0
                move.w  (a1,d0.w),10(a0)

                ;xx=c2*c1
                ;xy=c2*s1
                ;xz=s2
                ;yx=c3*s1+s3*s2*c1
                ;yy=-c3*c1+s3*s2*s1
                ;yz=-s3*c2
                ;zx=s3*s1-c3*s2*c1;s2*c1+c3*s1
                ;zy=-s3*c1-c3*s2*s1;c3*c1-s2*s1
                ;zz=c3*c2

                lea     Constants(pc),a1
                move.w  6(a0),d0
                move.w  (a0),d1
                move.w  d1,d2
                muls    d0,d1
                asr.l   #8,d1
                move.w  2(a0),d3
                muls    d3,d0
                asr.l   #8,d0
                move.w  d0,(a1)
                ;neg.w  d1
                move.w  d1,2(a1)
```

```
                move.w  4(a0),4(a1)
                move.w  8(a0),d4
                move.w  d4,d6
                muls    4(a0),d4
                asr.l   #8,d4
                move.w  d4,d5
                muls    d2,d5
                muls    10(a0),d2
                muls    d3,d4
                muls    10(a0),d3
                add.l   d4,d2
                sub.l   d5,d3
                asr.l   #8,d2
                asr.l   #8,d3
                move.w  d2,6(a1)
                neg.w   d3
                move.w  d3,8(a1)
                muls    6(a0),d6
                asr.l   #8,d6
                neg.w   d6
                move.w  d6,10(a1)
                move.w  10(a0),d0
                move.w  d0,d4
                muls    4(a0),d0
                asr.l   #8,d0
                move.w  d0,d1
                move.w  8(a0),d2
                move.w  d2,d3
                muls    (a0),d0
                muls    2(a0),d1
                muls    (a0),d2
                muls    2(a0),d3
                sub.l   d1,d2
                asr.l   #8,d2
                move.w  d2,12(a1)
                add.l   d0,d3
                asr.l   #8,d3
                neg.w   d3
                move.w  d3,14(a1)
                muls    6(a0),d4
                asr.l   #8,d4
                move.w  d4,16(a1)

                rts

Coses_Sines     dc.w    0,0,0,0,0,0
Angles          dc.w    0,0,0
Constants       dc.w    0,0,0,0,0,0,0,0,0

;Sintab is a table of 1024 sinus values with a radius of 256
;that I have further down my code...
```

A 2. A line drawing routine for filled vectors in assembler:

```
* written for kuma-seka ages ago, works fine and
* can be optimized for special cases...
* the line is (x0,y0)-(x1,y1) = (d0,d1)-(d2,d3) ...
* Remember that you must have DFF000 in a6 and
* The screen start address in a0.
* Only a1-a7 and d7 is left unchanged.
*    __    .
*  /( |( )|\/ '(|)
* /  )|(|\|/\   |)

Screen_widht=40 ;40 bytes wide screen...
fill_lines:     ;(a6=$dff000, a0=start of bitplane to draw in)

        cmp.w   d1,d3
        beq.s   noline
        ble.s   lin1
        exg     d1,d3
        exg     d0,d2
lin1:   sub.w   d2,d0
        move.w  d2,d5
        asr.w   #3,d2
        ext.l   d2
        sub.w   d3,d1
        muls    #Screen_Widht,d3        ;can be optimized here..
        add.l   d2,d3
        add.l   d3,a0
        and.w   #$f,d5
        move.w  d5,d2
        eor.b   #$f,d5
        ror.w   #4,d2
        or.w    #$0b4a,d2
        swap    d2
        tst.w   d0
        bmi.s   lin2
        cmp.w   d0,d1
        ble.s   lin3
        move.w  #$41,d2
        exg     d1,d0
        bra.s   lin6
lin3:   move.w  #$51,d2
        bra.s   lin6
lin2:   neg.w   d0
        cmp.w   d0,d1
        ble.s   lin4
        move.w  #$49,d2
        exg     d1,d0
        bra.s   lin6
lin4:   move.w  #$55,d2
lin6:   asl.w   #1,d1
        move.w  d1,d4
        move.w  d1,d3
        sub.w   d0,d3
        ble.s   lin5
        and.w   #$ffbf,d2
```

```
lin5:   move.w  d3,d1
        sub.w   d0,d3
        or.w    #2,d2
        lsl.w   #6,d0
        add.w   #$42,d0
bltwt:  btst    #6,2(a6)
        bne.s   bltwt
        bchg    d5,(a0)
        move.l  d2,$40(a6)
        move.l  #-1,$44(a6)
        move.l  a0,$48(a6)
        move.w  d1,$52(a6)
        move.l  a0,$54(a6)
        move.w  #Screen_Widht,$60(a6)   ;width
        move.w  d4,$62(a6)
        move.w  d3,$64(a6)
        move.w  #Screen_Widht,$66(a6)   ;width
        move.l  #-$8000,$72(a6)
        move.w  d0,$58(a6)
noline: rts
```

A 3. The quicksort in 68000 assembler
=====================================

```
* Sorts a list that looks like:
* Next entry offset.w, (x,y,z).w.
* all offsets must be set except for first entry's previous offset
* and the last entry's next offset.
* Offsets are FROM FIRST ADDRESS of sorting list
* a5=first address of sorting list!
*    __    .
*  /( |( )|\/ '(|)
* /  )|(|\|/\   |)


WghtOffs=6
NextOffs=0

QuickSort       ;(a5=start of sortlist,
                ; d0=0 (pointer to first entry, first time=0)
                ; d1=number of entries)


                cmp.w   #1,d1
                ble.s   .NothingToSort          ;don't sort if <=1 entries
                moveq   #0,d4                   ;size list 1
                moveq   #0,d5                   ;size list 2
                move.w  d0,d6                   ;first Nentry=d0

                move.w  WghtOffs(a5,d0.w),d2    ;d2=Pivot weight
                move.w  NextOffs(a5,d0.w),d3    ;d3=2nd entry
                subq.w  #2,d1                   ;Dbf-loop+skip first

.Permute        cmp.w  WghtOffs(a5,d3.w),d2     ;entry weight<pivot weight?
                ble.s   .Lower

                move.w  d6,NextOffs(a5,d3.w)    ;Insert BEFORE Nentry
                addq.w  #1,d4                   ;increase size of list 1
                move.w  d3,d6                   ;Set new Nentry

                bra.s   .Done                   ;Continue the loop...

.Lower          move.w  NextOffs(a5,d0.w),NextOffs(a5,d3.w)
                move.w  d3,NextOffs(a5,d0.w)    ;insert AFTER first entry
                addq.w  #1,d5                   ;size of list 2

.Done           move.w  NextOffs(a5,d3.w),d3    ;Get next entry
                dbf     d1,.permute

                move.w  d0,-(a7)                ;save Fentry..

                move.w  NextOffs(a5,d0.w),d0    ;Sort at entry after first
                move.w  d5,d1                   ;Size of list 2

                movem.w d4/d6,-(a7)             ;Save important registers
                bsr     QuickSort               ;and sort list 2
                movem.w (a7)+,d4/d6             ;d1 is now First Entry...
                move.w  (a7)+,d1
```

```
                move.w  d0,NextOffs(a5,d1.w)    ;Put first entry of
                                                ;list 2 after Fentry...
                move.w  d6,d0                   ;Sort at Nentry
                move.w  d4,d1                   ;size of list 1

                bsr     QuickSort               ;no important registers
                                                ;left...
.NothingToSort
                        ;Now the offset to the first entry is in d0!
                        ;to get the other values in the correct order
                        ;just go down the list (using nextoffs.)
                        ;First object is the heaviest...

                rts
```

A 4. The Insert Sort in 68020 assembler:
==

```
* This isn't exactly as the algorithm described earlier,
* it begins with creating a list and then stores the ADDRESSES of the
* sorted data in 2ndBUF instead...
* This sorts all lists, just specify offset to weight (word) and
* size of each entry. You don't need any pre-formatting.
* note that you HAVE TO change a line if you want this to work
* on 68000.. I've got a scaled index at one point. replace it
* with the lines after the semicolon.
*    __   .
*  /( |( )|\/ '(|)
* /  )|(|\|/\   |)

WghtOffs=4
EntrySize=6

InsertSort
                ;(a5=start of data
                ; a4=start of checklist
                ; a3=start of 2ndBUF
                ; d0 is lowest value of entries
                ; d1 is highest value
                ; d2 is number of entries

                movem.l a4/a5,-(a7)

                sub.w   d0,d1           ;max size of checklist this sort.
                subq.w  #1,d2
                subq.w  #1,d1           ;Dbf-loops...

                move.w  d1,d3           ;clear used entries
.ClearChecklist clr.w   (a4)+
                dbf     d3,.ClearCheckList

                move.w  d2,d3           ;transform...
.Transform      sub.w   d0,WghtOffs(a5)
                addq.w  #EntrySize,a5
                dbf     d3,.Transform

                movem.l        (a7),a4/a5

                move.w  d2,d3           ;Insert next line instead for
.AddisList      move.w  WghtOffs(a5),d0 ;68000 compatibility...
                addq.w  #4,(a5,d0.w*2)  ;add.w d0,d0 addq.w #4,(a5,d0.w)
                addq.w  #EntrySize,a5
                dbf     d3,.AddisList

                moveq   #-4,d0          ; #-lwdsize
.GetMemPos      add.w   d0,(a4)
                move.w  (a4)+,d0
                dbf     d1,.GetMemPos

                movem.l (a7)+,a4/a5
.PutNewList     move.w  WghtOffs(a5),d0
                move.w  (a4,d0.w),d0
```

```
move.l  a5,(a3,d0.w)
addq.w  #EntrySize,a5
dbf     d2,.PutNewList

;In this case you have a list of ADDRESSES to
;each object. I made it this way to
;make it more flexible (you maybe have more
;data in each entry than me?).

rts
```

Appendix B: Further Information

B 1: 'Proof' of hidden-plane elimination equation

I presented the following equation:
c=(x3-x1)*(y2-y1)-(x2-x1)*(y3-y1)
as a calculation of the normal vector of the plane
that the polygon in question spanned.

We had three points:
p1(x1,y1)
p2(x2,y2)
p3(x3,y3)

If we select p1 as base-point, we can construct the following
vectors of the rest of the points:

V1=(x3-x1,y3-y1,p)
V2=(x2-x1,y2-y1,q)

Where p and q in the z value denotes that we are not interested in
this value, but we must take it in our calculations anyway.
(These values are NOT the same as the original z-values
after the 2d->3d projection)

Now, we can get the normal vector of the plane that these vectors
span by a simple cross-product:

V1 x V2 =

```
  |  i       j      k|
= |(x3-x1) (x2-x1)  p|  (if i=(1,0,0), j=(0,1,0), k=(0,0,1))
  |(y3-y1) (y2-y1)  q|  (p and q are non-important)
```

But we are only interested in the Z-direction of the
result-vector of this operation, which is the same as
getting only the Z-coordinate out of the cross-product:

Z of (V1xV2) = (x3-x1)*(y2-y1)-(x2-x1)*(y3-y1)

Now if Z is positive, this means that the resultant vector
is aiming INTO the screen (positive z-values)
QED /Asterix

B 2. How to make a fill line out of the blitters line-drawing

You can't use the blitter line-drawing as it is and
draw lines around a polygon without a few special changes.

To make a fill-line routine out of a normal-lineroutine:

First, make sure it draws lines as it should,
many line-drawers I've seen draws lines to wrong points
Make sure you use Exclusive or- instead of or-minterm
Always draw lines DOWNWARDS. (or UPWARDS, if you prefer that)

Before drawing the line and before blit-check, eor the FIRST
POINT ON THE SCREEN THAT THE LINE WILL PASS.
Use fill-type line mode.

B 3: An alternate approach to rotations in 3-space by M. Vissers

/* This is a text supplied by Michael Vissers, and was a little
longer. I removed the part about projection from 3d->2d,
which was identical to parts of my other text in chapter 3.
If you know some basic linear algebra, this text might be
easier to melt than the longer version discussed in chapter 4.
If you didn't get how you were supposed to use the result in
chapter 4 then try this part instead. */

[ ] All you have to do is using these 3D matrices :

(A/B/G are Alpha,Beta and Gamma.) /* A,B,C = Angles of rotation */

```
|  cosA  -sinA  0  |    |  cosB   0  -sinB  |    |   1    0      0    |
|  sinA   cosA  0  |    |   0     1    0    |    |   0   cosG  -sinG  |
|   0      0    1  |    |  sinB   0   cosB  |    |   0   sinG   cosG  |
```

These are the rotation matrices around the x,y and z axis'. If you would
use these you'll get 12 muls'. 4 four for each axis. But, if you multiply
these three matrices with eachother you'll get only 9 muls'. Why 9 ???
Simple : after multiplying you'll get a 3x3 matrice, and 3*3=9 !

It doesn't matter if you do not know how to multiply these matrices. It's
not important here so I'll just give the 3x3 matrice after multiplying :

(c = cos, s = sin, A/B/G are Alpha,Beta and Gamma.)

```
        |     cA*cB              -cB*sA          sB   |
        | cG*sA-sB*cA*sG      cA*cG+sG*sA*sB     cB*sG |
        |-sG*sA-sB*cA*cG     -cA*sG+sA*sB*cG     cG*cB |
```

I hope I typed everything without errors :) Ok, how can we make some
coordinates using this matrice. Again, the trick is all in multiplying.
To get the new (x,y,x) we need the original points and multiply these with
the matrice. I'll work with a simplyfied matrice. (e.g. H = cA*cB etc...)

```
                      x   y   z   ( <= original coordinates)
                    -------------
        New X =     |   H   I   J   |
        New Y =     |   K   L   M   |
        New Z =     |   N   O   P   |
```

So...

```
        New X = x * H + y * I + z * J
        New Y = x * K + y * L + z * M
        New Z = x * N + y * O + z * P
```

Ha ! That's a lot more than 9 muls'. Well, actually not. To use the matrice
you'll have to precalculate the matrice.

Always rotate with your original points and store them somewhere else.
Just change the angles to the sintable to rotate the shape.
If you rotate the points rotated the previous frame you will lose all detail
until nothing is left.

So, every frame looks like this :
- pre calculate new matrice with given angles.
- Calculate points with stored matrice.

[ ]
The resulting points are relative to (0,0). So they can be negative to.
Just use a add to get it in the middle of the screen.

NOTE: Always use muls,divs,asl,asr etc. Data can be both positive and negative. Also, set the original coordinates as big as possible, and after rotating divide them again. This will improve the quality of the movement.

(Michael Vissers)

B 4: A little math hint for more accurate vector calculations

When doing a muls with a value and then downshifting the value, use
and 'addx' to get roundoff error instead of truncated error, for
example:

```
        moveq        #0,d7
DoMtxMul
        :
        muls        (a0),d0              ;Do a muls with a sin value *256
        asr.l        #8,d0
        addx.w        d7,d0               ;roundoff < trunc
        :
```

When you do a 'asr' the last outshifted bit goes to the x-flag.
if you use an addx with source=0 => dest=dest+'x-flag'.
This halves the error, and makes complicated vector objects
less 'hacky'.

```
 /)    __   .
((   /( |( )|\/  '(|)
 )) /  )|(|\|/\    |)
(/
```

********** 11. Interrupts **********

Legal Interrupt Handling for Amiga Demos

OS Legal interrupts are so easy to add and have such a very
tiny overhead that it is a real pity that so many people ignore
them...

Hardware interrupts are set with AddIntServer
and removed with RemIntServer .

The best way to show how to use interrupts on the Amiga is with
an example (thanks Bjorn!). I've pessimised it slightly (to make
it a little more readable) and fixed one bug!

```
INTB_VERTB  equ   5                 ; for vblank interrupt
INTB_COPER  equ   4                 ; for copper interrupt

CALL: MACRO
      jsr   _LVO\1(a6)
      ENDM


SystemOff:
      move.l   GfxBase,a6
      sub.l    a1,a1
      CALL     LoadView             ;Get default view
      CALL     WaitTOF
      CALL     WaitTOF
      CALL     OwnBlitter           ;Claim ownership of blitter
      move.l   $4.w,a6
      CALL     Forbid               ;Forbid multitasking
      moveq.l  #INTB_VERTB,d0       ; INTB_COPER for copper interrupt
      lea      VBlankServer(pc),a1
      CALL     AddIntServer         ;Add my interrupt to system list
      rts

SystemOn:
      move.l   $4.w,a6
      moveq.l  #INTB_VERTB,d0       ;Change for copper interrupt.
      lea      VBlankServer(pc),a1
      CALL     RemIntServer         ;Remove my interrupt
      CALL     Permit               ;Permit multitasking
      move.l   GfxBase,a6
      CALL     DisownBlitter        ;Give blitter back
      move.l   MyView,a1
      CALL     LoadView             ;Load original view
      rts

IntLevel3:

      movem.l  d2-d7/a2-a4,-(sp)    ; all other registers can be trashed
      ...
      movem.l  (sp)+,d2-d7/a2-a4

;     If you set your interrupt to priority 10 or higher then
;     a0 must point at $dff000 on exit.
```

```
      moveq    #0,d0                ; must set Z flag on exit!
      rts                           ;Not rte!!!

VBlankServer:
      dc.l  0,0                     ;ln_Succ,ln_Pred
      dc.b  2,0                     ;ln_Type,ln_Pri
      dc.l  IntName                 ;ln_Name
      dc.l  0,IntLevel3             ;is_Data,is_Code

IntName:dc.b "Dexion & SAE IntLevel3",0     ; :-)
      EVEN
;---------------------------------------------------------------------
```

where MyView is filled in immediately after graphics.library
have been opened:

```
   ..
   move.l   GfxBase,a1
   move.l   gb_ActiView(a1),MyView
   ...
```

********** 11.a Interrupts - AddIntServer() **********

NAME
AddIntServer -- add an interrupt server to a system server chain

SYNOPSIS
```
AddIntServer(intNum, interrupt)
  -168       D0-0:4      A1

void AddIntServer(ULONG, struct Interrupt *);
```

FUNCTION
This function adds a new interrupt server to a given server chain. The node is located on the chain in a priority dependent position. If this is the first server on a particular chain, interrupts will be enabled for that chain.

Each link in the chain will be called in priority order until the chain ends or one of the servers returns with the 68000's Z condition code clear (indicating non-zero). Servers on the chain should return with the Z flag clear if the interrupt was specifically for that server, and no one else. VERTB servers should always return Z set. (Take care with High Level Language servers, the language may not have a mechanism for reliably setting the Z flag on exit).

Servers are called with the following register conventions:

D0 - scratch
D1 - scratch

A0 - scratch
A1 - server is_Data pointer (scratch)

A5 - jump vector register (scratch)
A6 - scratch

all other registers must be preserved

INPUTS
intNum - the Paula interrupt bit number (0 through 14). Processor level seven interrupts (NMI) are encoded as intNum 15. The PORTS, COPER, VERTB, EXTER and NMI interrupts are set up as server chains.
interrupt - pointer to an Interrupt structure. By convention, the LN_NAME of the interrupt structure must point a descriptive string so that other users may identify who currently has control of the interrupt.

WARNING
Some compilers or assemblers may optimize code in unexpected ways, affecting the conditions codes returned from the function. Watch out for a "MOVEM" instruction (which does not affect the condition codes) turning into "MOVE" (which does).

BUGS
The graphics library's VBLANK server, and some user code, currently assume that address register A0 will contain a pointer to the custom chips. If you add a server at a priority of 10 or greater, you must compensate for this by providing the expected value ($DFF000).

********** 11.b Interrupts - RemIntServer() ***********

NAME
RemIntServer -- remove an interrupt server from a server chain

SYNOPSIS
```
RemIntServer(intNum, interrupt)
   -174        D0      A1

void RemIntServer(ULONG,struct Interrupt *);
```

FUNCTION
This function removes an interrupt server node from the given server chain.

If this server was the last one on this chain, interrupts for this chain are disabled.

INPUTS
intNum - the Paula interrupt bit (0..14)
interrupt - pointer to an interrupt server node

BUGS
Before V36 Kickstart, the feature that disables the interrupt would not function. For most server chains this does not cause a problem.

********** 12. Debugging ***********

Debugging your Code

A decent debugger is *essential* for anyone planning to program in 68000. If you haven't got a decent debugger, I suggest you get MonAm (free with Hisoft Devpac and Hisoft Hi-Speed Pascal) which is very very good. If you're using AsmOne there is a debugger implemented. Use ad instead of a when assembling. Step down with arrow right (allows you to jump to subroutines).
There are some other tips you can use:

1. Debugging with Enforcer
2. Breaking out of loops
3. AA Monitor problems

********** 12.1 Debugging - With Enforcer ***********

Debugging with Enforcer

Commodore have written a number of utilities that are *excellent* for debugging. They are great for trapping errors in code, such as illegal memory access and using memory not previously allocated.

The down side is they need to things:

a) A Memory Management Unit (at least for Enforcer). This rules out any 68000 machine, and (unfortunately) the Amiga 1200 and the Amiga 4000/EC030. If you are seriously into programming insist on a FULL 68030/40 chip, accept no substitute. Amiga 2000 owners on a tight budget may want to look at the Commodore A2620 card (14Mhz 68020 with 68851 MMU fitted) which will work and is now very cheap.

b) A serial terminal. This is really essential anyway, any serious programmer will have a terminal (I have an old Amiga 500 running NCOMM for this task) to output debug information with dprintf() from their code. This is the only sensible way to display debug info while messing with copperlists and hardware displays. If you can't afford a terminal, or you are very short of desk space, then another utility, called Sushi, will redirect this output to a file or window on your Amiga, although you will have to keep the system alive for this to work properly...

Enforcer, Mungwall and other utilities are available on Fred Fish Disks, amiga.physik and wuarchive, and probably on an issue of the excellent "The Source" coders magazine from Epsilon.

********** 12.2 Debugging - Breaking out of loops ***********

How to break out of never-ending loops

Another great tip for Boerge here:

This is a simple tip I have. I needed to be able to break out of my
code if I had neverending loops. I also needed to call my exit code when I did
this. Therefore I could not just exit from the keyboard interrupt which I have
taken over(along with the rest of the machine). My solution wa to enter
supervisor mode before I start my program, and if I set the stack back then
I can do an RTE in the interrupt and just return from the Supervisor() call.
This is snap'ed from my code:

```
   lea     .SupervisorCode,a5
   move.l  sp,a4           ;
   move.l  (sp),a3         ;
   EXEC    Supervisor
   bra     ReturnWithOS

.SupervisorCode
   move.l  sp,crashstack   ; remember SSP
   move.l  USP,a7          ; swap USP and SSP
   move.l  a3,-(sp)        ; push return address on stack
```

that last was needed because it was a subroutine that RTSes (boy did I have
porblems working out my crashes before I fixed that)
Then I have my exit code:

```
ReturnWithOS
   tst.l   crashstack
   beq     .nocrash
   move.l  crashstack,sp
   clr.l   crashstack
   RTE                     ; return from supervisor mode
.nocrash
```

my exit code goes on after this.

This made it possible to escape from an interrupt without having to care
for what the exception frames look like.

(CJ) I haven't tried this because my code never crashes. ;-)

********** 12.3 Debugging - AA Monitor Problems **********

Monitors

If you are using AA-chipmodes, or want to make your code
compatible with it, you must also make sure you code
works with every MONITOR on the market. Not only the
computer (Thanks to Alien/A poor group in Ankara
but not Bronx), for spotting this in my JoyRide2 Intro.

This is *extremely* dificult. See Monitor Problems
in the AGA chapter.

********** 13. Input **********

Handling Input for Amiga Demos

Keyboard Timings

If you have to read the keyboard by hardware, be very careful with your timings. Not only do different processor speeds affect the keyboard timings (for example, in the game F-15 II Strike Eagle on an Amiga 3000 the key repeat delay is ridiculously short, you ttyyppee lliikkee tthhiiss aallll tthhee ttiimmee. You use up an awful lot of Sidewinders very quickly!), but there are differences between different makes of keyboard, some Amiga 2000's came with Cherry keyboards, these have small function keys the same size as normal alphanumeric keys - these keyboards have different timings to the normal Mitsumi keyboards.

Use an input handler to read the keyboard. The Commodore guys have spent ages writing code to handle all the different possible hardware combinations around, why waste time reinventing the wheel?

********** 13.2 Input - Bogus Mouse Clicks **********

Beware bogus input falling through to Workbench

If you keep multitasking enabled and run your own copperlist remember that any input (mouse clicks, key presses, etc) fall through to the workbench. The correct way to get around this is to add an input handler to the IDCMP food chain (see - you *do* have to read the other manuals!) at a high priority to grab all input events before workbench/cli can get to them. You can then use this for your keyboard handler too (no more $bfexxx peeking, PLEASE!!!)

Look at the sourcecode for Protracker for an excellent example of how to do the job properly. Well done Lars!

********** 13.3 Input - Hardware Differences **********

Hardware Differences

The A1200 has a different keyboard to older Amigas. One of the side effects of this is it appears that older hardware-hitting keyboard read routines are not able to register more than one keypress at a time. I currently do not know whether this is a limitation in hardware and if it is possible to read multi-key presses (excepting the obvious CTRL/ALT/SHIFT type combinations) at all... A bit annoying for games writers I would think.

If you now are using you own hardware and Interrupts to read the keyboard on faster computers, make sure you ALWAYS have the given time-delay for ALL keyboards you want your program to work with (The delay between or.b #$40,$bfee01 and or.b #$40,$bfee01)! Don't trust delay loops since the cache can speed those up rather drastically!
I have seen too many, even commercial, programs that just ignores this and have NO delay code or just a simple dbf-loop. After about 15 keypresses your keyboard is dead and there is no code to reset it.
If you can - skip having your own keyboard routines, since they mostly fail anyway.

********** 14. Kickstart ***********

Important Kickstart Differences on Amiga

There are major differences between the early kickstart machines (Kickstart 1.2) and current releases (Kickstart 3.0/3.1)

Important differences:

Size: Kickstart 1.2 was 256Kb. Kickstart 3.0 is 512Kb

Offsets: *EVERYTHING* changes. Do not make any assumptions about any data in rom, for example reset locations, topaz.font data position.

Libraries: Many disk-based libraries under 1.3 are now in ROM, along with disk-validator and other things....

Workbench: Workbench is much improved. Use it.

OS Functions: *Many* new OS functions in all libraries. Now much easier to use, and faster. Much faster than under 1.2/1.3

How to check kickstart:

```
   move.l   4.w,a6
   move.l   LIB_VERSION(a6),d0
```

d0 now contains version number. Compare with the following (all values in decimal)

V0 to V32 - Obsolete! No longer supported.

V33 - Kickstart 1.2

V34 - Kickstart 1.3 (1.2 with autoboot for HD)

V35 - Early beta-kickstart 1.4. Obsolete

V36 - Obsolete! Early V2.00-V2.03 supplied with Amiga 3000
Amiga 3000 owners should upgrade to at least V37

V37 - Kickstart 2.04. Final release version of Kickstart 2

(V38) - Workbench 2.1 (exec.library should not show this version. All true V38 libraries are disk based)

V39 - Kickstart 3.0

V40 - Kickstart 3.1 - Currently only in AmigaCD 32

Do NOT compare numbers directly, eg.

```
   cmp.w #39,d0
```

```
   beq   kickstart3
```

Always check for greater or equal... eg.

```
   cmp.w #39,d0
   bcc   kickstart3     (bcc = BHIS, branch higher or same, unsigned)
```

********** 15. Misc **********

Miscellaneous hints and tips

1 How to make a RESET
2 Trackloaders

********** 15.1 Misc - How to make a Reset **********

How to make a RESET

Here is the official routine supported by Commodore:
                           ^^^^^^^^^^^^^^^^^^^^^^^^

```
  INCLUDE "exec/types.i"
  INCLUDE "exec/libraries.i"

;  csect text                  ; in lattice ASM
;  section text,code           ; in Devpac/Argasm

  xdef  _ColdReboot
  xref  _LVOSupervisor

EXECBASE        equ 4
ROMEND          equ $01000000
SIZE_OFFSET     equ -$14

KICK_V36        equ 36
V36_ColdReboot  equ -726


_ColdReboot:
      move.l EXECBASE,a6
      cmp.w  #KICK_V36,LIB_VERSION(a6)   ;which Version of Exec ?
      blt.s  .old_kick                   ;old one -> goto old_kick

      jmp    V36_ColdReboot(a6)          ;else use Exec-Function

.old_kick:
      lea    .Reset_Code(pc),a5
      jsr    _LVOSupervisor(a6)          ;get Supervisor-status
      ;never reaching this point

      cnop 0,4                             ;very important
.Reset_Code:
      lea    ROMEND,a0                   ;Calc Entrypoint
      sub.l  SIZE_OFFSET(a0),a0
      move.l 4(a0),a0
      subq.l #2,a0
      reset                              ;Reset peripherie
      jmp    (a0)                        ;done
                  ; <reset> and <jmp> in the same LONGWORD !!!!
      END
```

********** 15.2 Misc - Trackloaders **********

Trackloaders

Use CIA timers! DON'T use processor timing. If you use processor timing you will MESS UP the diskdrives in accelerated Amigas.

Use AddICRVector to allocate your timers, don't hit $bfxxxx addresses!!!

On second thoughts. DON'T use trackloaders! Use Dos...

Optimising your Code

Everyone wants their code to run as fast as possible, so here are some speed-up tricks for you:

1 68000 Optimisation
2 68020 Optimisation - See 680x0 chapter
3 Blitter Speed Optimisation - see blitter chapter
4 General Speed-Up Notes

68000 optimisation

Written by Irmen de Jong, march '93. (E-mail: ijdjong@cs.vu.nl)
Some notes added by CJ

NOTE! Not all these optimisations can be automatically applied. Make sure they will not affect other areas in your code!

Original	Possible optimisation	Examples/notes
STANDARD WELL-KNOWN optimisATIONS		
RULE: use Quick-type/Short branch! Use INLINE subroutines if they are small!		
BRA/BSR xx	BRA.s/BSR.s xx	if xx is close to PC
MOVE.X #0	CLR.X/MOVEQ/SUBA.X	move.l #0,count -> clr.l count
move.l #0,d0	-> moveq #0,d0	
move.l #0,a0	-> sub.l a0,a0	
CLR.L Dx	MOVEQ #0,Dx	
CMP #0	TST	
MOVE.L #nn,dx	MOVEQ #nn,dx	possible if -128<=nn<=127
ADD.X #nn	ADDQ.X #nn	possible if 1<=nn<=8
SUB.X #nn	SUBQ.X #nn	same...
JMP/JSR xx	BRA/BSR xx	possible if xx is close to PC
JSR xx;RTS	JMP xx	save a RTS
BSR xx;RTS	BRA xx	same... (assuming routine doesn't rely on anything in the stack)
LSL/ASL #1/2,xx	ADD xx,xx [ADD xx,xx]	lsl #2,d0 -> 2 times add d0,d0
MULU #yy,xx where yy is a power of 2, 2..256		
LSL/ASL #1-8,xx		mulu #2,d0 -> asl #1,d0 -> add d0,d0
BEWARE: STATUS FLAGS ARE "WRONG"		
DIVU #yy,xx where yy is a power of 2, 2..256		
LSR/ASR #.. SWAP		divu #16,d0 -> lsr #4,d0
BEWARE: STATUS FLAGS ARE "WRONG", AND HIGHWORD IS NOT THE REMAINDER.		
ADDRESS-RELATED OPTIMISATIONS		
RULE: use short adressing/quick adds!		
MOVEA.L #nn	MOVEA.W #nn	Movea is "sign-extending" thus possible if 0<=nn<=$7fff
ADDA.X #nn	LEA nn()	adda.l #800,a0 -> lea 800(a0),a0

```
                              possible if -$8000<=nn<=$7fff

LEA nn()        ADDQ.W #nn    lea 6(a0),a0 -> addq.w #6,a0
                              possible if 1<=nn<=8

$0000nnnn.l     $nnnn.w       move.l   4,a6 -> move.l 4.w,a6
                              possible if 0<=nnnn<=$7fff
                              (nnnn is SIGN EXTENDED to LONG!)

MOVE.L #xx,Ay   LEA xx,Ay     try xx(PC) with the LEA

MOVE.L Ax,Ay;
ADD #nnnn,Ay    LEA nnnn(Ax),Ay   copy&add in one

OFFSET-RELATED OPTIMISATIONS
RULE: use PC-relative addressing or basereg addressing!
      put your code&data in ONE segment if possible!
-------------------------------------------------------------------------
MOVE.X nnnn     MOVE.X nnnn(pc)      lea copper,a0 -> lea copper(pc),a0..
LEA nnnn        LEA nnnn(pc)      ...possible if nnnn is close to PC

(Ax,Dx.l)       (Ax,Dx.w)            possible if 0<=Dx<=$7fff

If PC-relative doesn't work, use Ax as a pointer to your data block.
Use indirect addressing to get to your data: move.l Data1-Base(Ax),Dx etc.

TRICKY OPTIMISATIONS
-------------------------------------------------------------------------
BSET #xx,yy     ORI.W #2^xx,yy       0<=xx<=15
BCLR #xx,yy     ANDI.W #~(2^xx),yy      "
BCHG #xx,yy     EORI.W #2^xx,yy         "
BTST #xx,yy     ANDI.W #2^xx,yy         "
                Best improvement if yy=a data reg.
                BEWARE: STATUS FLAGS ARE "WRONG".

SILLY OPTIMISATIONS (FOR OPTIMISING COMPILER OUTPUTS ETC)
-------------------------------------------------------------------------
MOVEM (one reg.)  MOVE.l         movem  d0,-(sp) -> move.l d0,-(sp)

MOVE xx,-(sp)     PEA xx         possible if xx=(Ax) or constant.

0(Ax)             (Ax)

MULU/MULS #0      CLR.L          moveq #0,Dx with data-registers.

MULU #1,xx        SWAP CLR SWAP  high word is cleared with mulu #1
MULS #1,xx        SWAP CLR SWAP EXT.L  see MULU, and sign exteded.
                                  BEWARE: STATUS FLAGS ARE "WRONG"

LOOP OPTIMISATION.
-------------------------------------------------------------------------
Example: imagine you want to eor 4096 bytes beginning at (a0).
Solution one:

      move.w   #4096-1,d7
.1    eori.b   d0,(a0)+
      dbra     d7,.1

Consider the loop from above. 4096 times a eor.b and a dbra takes time.
What do you think about this:
```

```
      move.w   #4096/4-1,d7
.1    eor.l    d0,(a0)+    ; d0 contains byte repeated 4 times
      dbra     d7,.1
```

Eors 4096 bytes too! But only needs 1024 eor.l/dbras.
Yeah, I hear you smart guys cry: what about 1024 eor.l without any loop?!
Right, that IS the fastest solution, but is VERY memory consuming (2 Kb).
Instead, join a loop and a few eor.l:

```
      move     #4096/4/4-1,d7
.1    eor.l    d0,(a0)+
      eor.l    d0,(a0)+
      eor.l    d0,(a0)+
      eor.l    d0,(a0)+
      dbra     d7,.1
```

This is faster than the loop before. I think about 8 or 16 eor.l's is just fine, depending on the size of the mem to be handled (and the wanted speed!). Also, mind the cache on 68020+ processors, the loop code must be small enough to fit in it for highest speeds.
Try to do as much as possible within one loop (but considering the text above) instead of a few loops after each other.

MEMORY CLEARING/FILLING.

A common problem is how to clear or fill some memory in a short time.
If it is CHIP-MEMORY, use the blitter (only D-channel, see below). In this case you can still do other things with your 680x0 while the blitter is busy erasing. If it is FAST-MEMORY, you can use the method from above, with clr.l instead of eor.l, but there is a much faster way:

```
   move.l   sp,TempSp
   lea      MemEnd,sp
   moveq    #0,d0
;...for all 7 data regs...
   moveq    #0,d7
   move.l   d0,a0
;...for 6 address regs...
   move.l   d0,a6
```

After this, ONE instruction can clear 60 bytes of memory (15*4):

```
   movem.l  d0-d7/a0-a6,-(sp)    ;wham!
```

Now, repeat this instruction as often as required to erase the memory. (memsize/60 times). You may need an additional movem.l to erase the last few bytes. Get sp(=a7) back at the end with (guess..):

```
   move.l   TempSp,sp
```

If you are low on mem, put a few movem.l in a loop. But, now you need a loop-counter register, so you'll only clear 56 bytes in one movem.l.

In the case of CHIP memory, you can use both the blitter and the processor simultaneously to clear much CHIP mem in a VERY short time...
It takes some experimentation to find the best sizes to clear with the blitter and with the processor.

BUT, ALWAYS USE A WaitBlit() AFTER CLEARING SIMULTANEOUSLY, even if you

think you know that the blitter is finished before your processor is.

********** 16.4 Optimizing - General Speed-up Notes ***********

- When optimising programs first try to find the time-critical parts (inner loops, interrupt code, often called procedures etc.) In most cases 10% of the code is responsible for 90% of the execution time. Don't waste time doing needless optimising on startup and exit code when it's only called once!

- Often it is better not to set BLTPRI in DMACON (#10 in $dff09a) as this can keep your processor from calculating things while the blitter is busy.

- Use as many registers as possible! Store values in registers rather than in memory, it's much faster!

- DON'T put your parameters on the stack before calling a routine! Instead, put them in registers!

- If you have enough memory, try to remove as many MULU/S and DIVU/S as possible by pre-calculating a multiplication or division table, and reading values from it, or rewrite multiply/divide code with simpler instructions if possible (eg ADD, LSR, etc.)

********** 17. Reading C ***********

READING C FOR ASSEMBLER PROGRAMMERS

This file has been left intact as I received it for HTC (apart
from AmigaGuide Conversion) There are a few errors in it, but in general
it's pretty good. The example programs referred to are in source/codemanual/
(but don't expect them to work!) - CJ

----------The EnConn BBS Better Programming Guide=======----------

Introduction: Read C, write assembler?

********** 17.0a Reading C - EnConn BBS **********

Call EnConn, The Engineering Connection on +612 524-1584
ASM coding our speciality. C coders welcome.
Home of coders such as:
 Piranha, Coz, BigMac, Drizzt, Questa, Punisher, Wildfire, Ziggy.
Enconn BBS is Real-Names-Only and Legal-Only .

Running MaxsBBS V1.52 Enhanced :)

Fidonet # 3:712/613
AmigaNet # 41:200/584
MaxsNet # 3:30000/99

********** 17.0b Reading C - Introduction ***********

INTRODUCTION

Some of the most useful utilities have come from the demo coders scene. Mostly they serve the purpose they were designed for but lack one thing. A graphical user interface or even just a friendly interface. How many ST module rippers have you seen that just use keyboard input from a cli window? Lame aren't they! All this builtin code going to waste. Instead they get out their Abacus machine language book & just copy the asm code to get keyboard input in a simple console window! Totally lame.

Ok, heres the reason.. Where else is there for an asm coder to get some examples!! Nowhere much. Theres all those ROM KERNAL MANUALS but all the examples are in bloody C code (also known as `line noise'.) But there are HEAPS of them in there. If only you could read them & learn how easy it is to convert these into assembler. Well I started reading the C examples and after a while you get to know exactly what is going on & can easily convert it to assembler. Just treat the code as example pseudo-code!
Please note, I had NEVER read any C books when I worked out how to convert it so don't be afraid to give it a go. I think the fact that I learned it this way makes me a pretty good choice for writing this article. To all the C coders that are likely to pick up a lot of mistakes.. Well please do but just remember, this is all the knowledge I ever need to READ C, WRITE ASSEMBLER.
I gave this text to a certain Asm coder who remains anonymous, he has now started writing system friendly code & reading the RKM's without any of the previous fears!! 8-)

==========
PLEASE NOTE: After writing this text I asked Andrew Patterson to read it & correct any OBVIOUS mistakes. Andrew is a competant C and Asm programmer unlike myself (I only know Asm really). His additions are marked with a "!" character at the begining of the line.
Thanks Andrew!!
==========

********** 17.1 Reading C - Braces and Brackets ***********

CHAPTER #1   BRACES and BRACKETS

OK, first off lets take a look at what these things {} (BRACES) do.

```
/*********************C-CODE******* LINE NUMBERS USED ONLY FOR DEMONSTRATION
1       {
2       intbase = OpenLibrary("intuition.library",0)
3       }
************************************/
```

Easy! its just a routine to open a library.
They just mark the start & end!!!

This would convert to:

```
;--------------------ASM-CODE----------
GetIB:
11      lea     intname,a0
12      moveq   #0,d0
13      movea.l (4).w,a6
14      jsr     _LVOOpenLibrary(a6)
15      move.l  d0,intbase

intname:
        dc.b    'intuition.library',0
        even
;---------------------------------------
```

Line1 is the start of the routine.
Line2 says "Open the intuition library & put the library base in "intbase"
Sometimes its easier to read the lines back to front to make sense of them.
Ok, look up OpenLibrary, it will tell you this:
 OpenLibrary(libname,version)(a0,d0) and that the result is returned in D0.
Meaning that it wants the address of the text "intuition.library" in A0 & a
version number in D0.
Line11 we put the address into A0
Line12 we tell it version 0 (ANY version)
Line13 OpenLibrary is an exec routine. So put execbase in A6.
Line14 we call the OpenLibrary routine
Line15 we store the result in intbase just like the C code

Most of the time you will see this though

```
/****************************************/
1    {
2    if (intbase = OpenLibrary("intuition.library",0))
3               {
4               if (wd = OpenWindow(newwin))
5                       {
6                       Do tricky stuff etc. etc.
7
8                       CloseWindow(newwin)
9                       }
10              CloseLibrary(intbase)
11              }
12
13   }
```

```
/****************************************/
```
Line2 says "Open intuition library, any version, store it in intbase"
The IF means "if we get it continue with the code in the next braces, otherwise, skip past the code in braces (to line 12)" The IF works on the statement within the brackets. It could just say IF (A = B).
If we got intbase, go to the code starting at Line3. It does the same thing except it is opening a window. If it gets the window, it continues at Line5 and if not, it skips past the matching brace where it will close the intbase (Line10) and then exit. Notice how it always skips to the next matching brace.
Here it is in Assembler:
```
;-------------------------------------------
Start:                          ;{
        movea.l (4).w,a6
        lea     intbase,a0
        moveq   #0,d0
        jsr     _LVOOpenLibrary(a6)
        move.l  d0,intbase      ;if (intbase = OpenLibrary("intuition.library",0))
        beq     exit
OpenWd:                         ;   {
        move.l  d0,a6
        lea     newwin,a0
        jsr     _LVOOpenWindow(a6)
        move.l  d0,wd           ;   if (wd = OpenWindow(newwin))
        beq     CloseInt
                                ;      {
        ;tricky stuff                   tricky stuff
        ;etc. etc.                      etc. etc.

CloseWin:
        move.l  intbase,a6
        move.l  newwin,a0
        jsr     _LVOCloseWindow(a6)     ;CloseWindow(newwin)
                                ;      }

CloseInt:
        movea.l (4).w,a6
        move.l  intbase,a1
        jsr     _LVOCloseLibrary(a6)    ;CloseLibrary(intbase)
                                ;   }
exit:
        moveq   #0,d0
        rts                     ;}
;-----------------------------------------------
```

Brackets ()
These are use to pass arguments to fuctions. Such as library routines or our own subroutines. Heres how they pass arguments.
/* Here i made up a routine called multiply that takes 2 numbers & multiplies them & then returns */

```
/*********************************************************/
1  int i1 = 2;
2  int i2 = 5;
3  ulong leftedge = 0;
4
5  leftedge = multiply(i1,i2); /* call the subroutine */
6  etc. etc. exit.
7/* a subroutine goes like this..  Return-value-type name(parameters)*/
8/*So this one returns an integer & wants 2 parameters
9
```

```
10int multiply(int m1, int m2)
11    {
12       int r;  /*set aside an integer sized variable*/
13
14       r = m1 * m2; /* multiply them*/
15       return r;    /* return the result */
16    }
/*********************************************************/
```

Line 1 makes an integer sized variable called i1 and gives it the value "2"
Line 2 & 3 do the same things
Line 5 calles the subroutine "multiply" giving it the 2 parameters "i2 & i1" and puts the result in the variable called "leftedge"
Line10 is our subroutine. It takes the 2 parameters & now puts them in new variable names (m1 & m2)
Line12 gets a variable for the result
Line14 does the big multiply
Line15 sets the return result & then RTS's

In Assembler. This is a pretty simple one. A trickier one might have a 32bit multiply routine (for the 68000) here.

```
;--------------------------------
        move.w  #2,d0
        move.w  #5,d1
        bsr     multiply
        move.w  d1,leftedge
        etc..

multiply:       ;value1 in d0,  value2 in d1,  result in d1
        mulu.w  d0,d1
        rts

leftedge dc.w 0

;---------------------------------------------------------------
```

If the subroutine says this: void multiply(int m1, int m2)
The word VOID just means that it does not return any value.

==========

Square Brackets.[] These normally signify an array. You might see this

UBYTE data[10]; This means define an array of 10 unsigned bytes.

Which is the same as

```
data:
        dcb.b   10,0
```

The data in the array can then be accessed like this:

```
  i = 2;
  data[i] = 50  /* moves 50 into the data+2 */
```

which is like

```
 moveq  #2,d0
 lea    data,a0
 move.b #50,0(a0,d0.w) ;moves 50 into data+2
```

or just

```
               data[2] = 50
```

converts to

```
lea data,a0
move.b  #50,2(a0)
```

********** 17.2 Reading C - Define, UBYTE, USHORT, etc. **********

CHAPTER #2 DEFINE (#define), UBYTE, USHORT etc

These are easy.

```
#define LEFT 10   is the same as   LEFT = 10
UBYTE       means an unsigned byte (like dc.b 0)
USHORT      means an unsigned word (like dc.w 0)
ULONG       means an unsigned long (like dc.l 0)
```

If they don't have the U (BYTE SHORT LONG) then they are using the most signifigant bit as the sign bit. A UBYTE can be any value from 0 - 255 but a signed BYTE can be -127 to +127.

INT This is just an integer. IMPORTANT! Some compilers treat INT as a word, others treat INT as a longword.

APTR this is A POINTER. The contents points to somewhere else.
Its like this.

```
screenptr:
          dc.l myscreen
......

myscreen:
        dc.w 10,10,12,etc
```

* A star in front of a variable name means its a pointer as well

```
!-----------------------------------------
!  A star in front of a pointer (ie something that is already defined as
!  a pointer) refers to what the pointer is pointing at.
!
!  ie char *string;     // Defines 'string' as the address of some characters
!
!  *string now means the character that 'string' points at.
!  myfunctionthatwantsacharacter(*string);
!
!  move.l  string,a0            ; a0 has the string address
!  move.b  (a0),d0              ; d0 has the character ie *string
!  jsr myfunctionthatwantsa..

!-----------------------------------------
!
! an & in front of a variable name means give me the address of this item
! ie
!       wd = OpenWindow(&newwindow);
!
! means like
!
!       lea newwindow,a0
!       .. call OpenWindow etc
!
! newwindow:
!       dcb.b   the size of a newwindow struct
```

BPTR this is a bcpl pointer. These are supposedly left over from a long lost language called BCPL which was used to code the dos.library before WB2.

BPTRs have to be multiplied (when reading) or divided (writing) by 4.

eg.

```
   move.l ThisTask,a0
   move.l pr_CurrentDir(a0),a0 ;prCurrentDir is a BPTR
   add.l  a0,a0
   add.l  a0,a0 ;we have to multiply it by 4 to get the address it points to
```

STRUCT This normally defines a preset structure & fills in the variables accordingly. Such as this:

```
struct Gadget mygad =
 {
 0,10,10,50,12,0,0,0,0,0,"Mygad",0,0,0
 }
```

OK, the C compiler sees this & looks up what a Gadget structure looks like.

```
        STRUCT  Gadget
          {
        struct Gadget *NextGadget;
        WORD    LeftEdge, TopEdge;
        WORD    Width, Height;
        UWORD   Flags;
        UWORD   Activation;
        UWORD   GadgetType;
        APTR    GadgetRender:
        APTR    SelectRender;
        struct  IntuiText *GadgetText;
        LONG    MutualExclude;
        APTR    SpecialInfo;
        WORD    GadgetID;
        APTR    UserData;
          }
```

```
struct Gadget mygad =
 {
 0,10,10,50,12,0,0,0,0,0,"Mygad",0,0,0,0
 }
```

One by one it fills it in like this.

```
mygad:
struct Gadget *NextGadget;      dc.l 0       ;a pointer to the next gadget
WORD    LeftEdge, TopEdge;      dc.w 10,10   ;left top
WORD    Width, Height;          dc.w 50,12   ;width height
UWORD   Flags;                  dc.w 0       ;flags
UWORD   Activation;             dc.w 0       ;activation
UWORD   GadgetType;             dc.w 0       ;gadgettype
APTR    GadgetRender:           dc.l 0       ;gadgetrender (none)
APTR    SelectRender            dc.l 0       ;selectrender
struct  IntuiText *GadgetText;  dc.l Mygad.txt ;gadget text
LONG    MutualExclude;          dc.l 0
APTR    SpecialInfo;            dc.l 0
WORD    GadgetID;               dc.w 0
APTR    UserData;               dc.l 0
```

The GadText one is special. It wants a pointer to the text. The C coder just

puts "MyGad". We have to do more than that. We put the address of our text in (as it wants *GadgetText (Pointer to our text)) but it also says "struct Intuitext" This tells the C compiler to create an intuitext structure & put its address here. So you have to look up what an intuitext stucture looks like, create it, and put its address in there.

It looks like this:

```
MyGad.txt
        dc.b 1,0,0,0   ;frpen,backpen,drawmode,align
        dc.w 10,10     ;left top
        dc.l 0         ;point to font
        dc.l gadtext   ;point to text
        dc.l 0         ;nexttext

gadtext dc.b 'MyGad',0
        even
```

So you can see that this is the sort of thing that saves the C coder a bit of time, we have to do a bit more work than they do here.

********** 17.3 Reading C - << >> = == ! || | && etc **********

CHAPTER #3 << >> = == ! || | && > < . + ++ +=

Math symbols

```
=  equals            a = 2         move.l #2,d0
+  add               a = a + 1     addq.l #1,d0
*  multiply          a = a * 5     mulu.w #5,d0
/  divide            a = a / 2     divu.w #2,d0
%  modulo            a = a % 3     divu.w #3,d0
                                   swap d0

|  OR                a = a | 2     OR.L #2,d0
&  AND               a = a & 1     AND.L #1,d0
^  EOR               a = a ^ 1     EOR.L #1,d0
~  NOT               a = ~a        NOT.L d0
<< ASL               a = a << 2    ASL.L #2,d0
>> ASR               a = a >> 2    ASR.L #2,d0

!----------------------------------------
! These can all be abbreviated if the variable to the left is the same as
! the variable to the right.
!
! ie  a += 1 is equivalent to a = a+1
!     a |= 2 is equivalent to a = a|2
!

eg.   Wait((1L<<win->userport->mp_sigbit));

Converts to..
        movea.l (4).w,a6   ;execbase
        movea.l win,a0     ;get our window
        movea.l wd_UserPort(a0),a0  ;get our windows userport
        movea.l MP_SIGBIT(a0),d1    ;get our windows userports signal bit no.
        moveq   #1,d0
        asl.l   d1,d0               ;Shift it left into its position.
        jsr     _LVOWait(a6)        ;Wait for the signal to arrive.
```

Conditional code

```
== Is exactly equal. They use this in C when they compare something.
eg.   if(a == 2)      cmp.l #2,d0
         {            beq  is2
         etc.   not2:
         }      is2:

In english, IF a is exactly equal to 2 then continue.

!= is NOT equal     eg.  if(a != 2) cmp.l #2,d0
                            { }     bne  not2
                               is2:
                               not2:
> < >= =<  greater than, less than, less or equal, greater or equal.
            BGT           BLT        BLE            BGE
```

? Seems to mean if true do this, if false do that.

such as   b = (a == 2) ? 10 : 0;

its saying.. if a is exactly 2 then b = 10. If not, b = 0

```
        cmp.l  #2,d0   ;is d0 2
        beq    do0     ;if so then d1 is 0
        move.l #10,d1  ;otherwise d1 is 10
        bra    done
do0:    move.l #0,d1
done:
```

&&   Think of this as being the same as AND in english!!

```
 if (a = 2 && b = 4)
```

will continue if both the statements are true

|| Think of this as being the same as OR in english!!

```
 if (a = 2 || b = 4)
```

will continue if either of the statements are true

Misc
====

++ = Increment   a++ means add 1 to a addq.l #1,d0 or move.w (a0)+,d0
-- = Decrement   a-- means subtract 1 subq.l #1,d0 or move.w -(a0),d0
8) a standard smiley

/* and */ These mark the start & end of a comment. Anything inbetween is ignored.

```
!--------------------------------------
! Some C compilers also allow C++ style comments. If a '//' is
! encountered the rest of the line is ignored
!
! wd = OpenWindow(&window);      // Window opens
```

CHAPTER #4 GETOFFSET (->), INDEX (.), ADDRESS (&)

Lets go straight to an example

```
/*********************************************/
1#define LEFT 10
2#define TOP 12
3
4       {
5       PrintIText(win->RPort,&myItext,LEFT,TOP)
6       etc.
7       etc.
8       }

/*********************************************/
```

Looking up PrintIText it says: PrintIText(rp,iText,left,top)
(a0, a1 ,d0 ,d1)
Rastport in A0, Intuitext in A1, Left & top edges in D0 & D1.

Line 5 just does those 4 things. But lets look at how its doing it

The 1st one, (A0) says win->RPort in C. In english it means get the contents of that offset in the array.

win is our window structure which we got from OpenWindow
It looks like this:

```
Window
    rsreset
    rs.l wd_NextWindow          ; for the linked list of a Screen
    rs.w wd_LeftEdge            ; screen dimensions
    rs.w wd_TopEdge             ; screen dimensions
    rs.w wd_Width               ; screen dimensions
    etc. etc.
    rs.l wd_MenuStrip           ; first in a list of menu headers
    rs.l wd_Title               ; title text for the Window
    rs.l wd_FirstRequest        ; first in linked list of active Requesters
    etc.  etc. etc.
    rs.l wd_RPort               ; this Window's very own RastPort
        and so on....
```

So what (win->RPort) is doing is this in assembler..

```
        movea.l win,a0  ;window in a0
        movea.l wd_RPort(a0),a0 ; like 50(a0) approximately
```

It gets the pointer from wd_RPort(a0) & puts it in the register

So now A0 has our rastport just where PrintIText wants it.

Next it says.. "&myItext" for A1. The & sign means ADDRESS

Which is just like writing.. lea myIText,a1

next it says LEFT,TOP for D0 & D1 which is the same as

```
        move.l  #LEFT,d0
        move.l  #TOP,d1
```

So the full thing in assembler is

```
;--------------PrintIText(win->RPort,&myItext,LEFT,TOP)
LEFT = 10
TOP = 12

        movea.l win,a0
        move.l  wd_RPort(a0),a0
        lea     myIText,a1
        move.l  #LEFT,d0
        move.l  #TOP,d1
        movea.l intbase,a6
        jsr     _LVOPrintIText(a6)

;-----------------------------------------
```

The other one to watch is the INDEX symbol (.)

You will see code like this at times.

```
        win.RPort
```

This works similar to

```
        win->RPort
```

but heres where its different

```
win->RPort will do this  movea.l win,a0
                         movea.l wd_RPort(a0),a0

win.RPort  will do this  movea.l win,a0
                         lea     wd_Rport(a0),a0

or                       movea.l win,a0
                         addi.l #wd_RPort,a0
```

It just points us to that location in the window structure while the GETOFFSET one (->) gives us the contents of that location.

You should take care not to confuse the two of these because nothing will work if you get them wrong.

********** 17.5 Reading C - loops **********

CHAPTER #5 Loops

There a few different C loops

The WHILE loop

```
 while (a < 10)
        {
        printf "Hello World"
        a = a + 1
        }
```

It executes the code in the braces WHILE a is less than 10. When a is 10 or more, it skips past the braces.

The FOR loop

```
 for (depth=0; depth<3; depth++)
  {
  bitmap.planes[depth] = 0
  etc
  }
```

Read it in english like this:

Set the start variable to zero, if depth is less than 3, execute the code in braces, then increment depth and loop back to depth<3.

So the 1st bit sets a variable, the 2nd bit is the conditions of the loop & the 3rd bit increments the starting variable.

********** 17.6 Reading C - Which INCLUDE files to include **********

CHAPTER #6 Which INCLUDE file to include?

Well here's the basic idea if you're trying to convert some C code. Just include the same ones they did but forget it if it says stuff like protos.h etc. Don't worry about them. I don't know or care what they are. When you start extending your code just have a look at what your working on. If you need something like a rastport structure then find the include file that has it in there & just include it. The best thing to do is get a truly kick-ass assembler like Macro68 & forget about include files forever. With this assembler you just assemble ALL your includes together into a library which the assembler uses. The file stays in memory so assembly is extremely fast! We did one assembly in Macro68 on BigMacs 040. It took 3 SECS for a 900k source file. Devpac3 took 2 MINUTES!!!!! NO JOKE!!!

********** 17.7 Reading C - What if C uses amiga.lib routines? ***********

CHAPTER #7   What if the C code uses amiga.lib routines!?

This is a real PAIN IN THE ASS!! What happens is that amiga.lib contains some ready made routines that sometimes just didn't make it into ROM. Then in the examples in the RKMs they use these "amiga.lib MACROS" without any mention of how to do the same thing in assembler. What you have to do is find your RKM(includes & autodocs) and look up AMIGA.LIB in the linker libraries section. Here you'll find the source code for each routine (IN C or ASM)!!! If its C its up to you to convert this back to assembler or at least work out what happens.

Another one is a C macro that just appears in the include files (such as graphics/gfxmacros.h)

Heres one of the macros in gfxmacros.h

```
#define ON_VBLANK       custom.intena = BITSET|INTF_VERTB;
```

This macro Turns on the VBlank interrupt

In assembler its..

```
    lea _custom,a5
    move.w #BITSET|INTF_VERTB,intena(a5)
```

or in democoders asm

```
    move.w #$8020,$dff09a
```

********** 17.8 Reading C - Complete example - bitmap scaling **********

CHAPTER #8 Complete example BITMAP SCALING USING THE OS

**********THIS CODE REQUIRES 2.04+ TO RUN**********
Two source codes are included with this text. Scale.C & Scale.asm

Both written by ME.Please forgive my C!! 8) Heres whats going on.

First, the includes. We just include all the same ones. Plus we need the
_lib.i includes for gfx, dos & intuition because the have the vales for
_LVOOpenWindow etc. etc. C compilers know them all I think or use those .fd
files.
!--
! Thats what the pragma and proto files do.
!

Next it defines the variables Intuitionbase & Gfxbase. They have to be
those longer names for the C compiler. It tells them that they are
pointers (*)

Next it goes straight into void main(). This generally is the standard
startup code which opens dos.library & handles WB loading etc. Since this is a
CLI only utility all I'll do is open the dos.library.
(for a demo or something you should include standard startup code.
 See the howtocode.txt for example source)

Then it defines a few other things like a taglist for openwindow & a bitmap
structure etc. In assembler we prefer this stuff at the bottom (have a look.)

Next it opens the intuition library version 37.
If it gets that it opens the graphics library v37
If it gets that it opens the Window.
Next it Initializes a BitMap structure. What we have to do is to just look up
what a bitmap structure looks like & save some space for it in our data area
which we do with   bm: dc.b bm_SIZEOF.

Next   InitBitMap(&bm,3,400,200);
Looking at the InitBitmap call it wants the bitmap structure in A0
The number of planes in d0
and the width & height in d0 & d1.

```
 lea  MyBitMap,a0
 moveq #3,d0
 move.l #400,d1
 move.l #200,d2
```

 Next it initializes a structure called BitScaleArgs. This can be found in
graphics/scale.i. What you have to do is set aside the bytes for it, fill in
the details, & then put the address in A0 & call BitMapScale.
 What we have planned is that our Bitmap that we have initialised with
InitBitMap will be scaled to the sizes specified and then copied to our
window's bitmap (The copy is done by BitMapScale). Since this is graphics
data & BitMapScale uses the BLITTER, our bitmaps planedata must be in
CHIP ram.

```
 Where it says:         bsa.SrcX = 0;
                        bsa.SrcY = 0;
```

```
                        bsa.SrcWidth = 400;
```

We convert to:

```
                        lea     bsa,a0
                        move.w  #0,bsa_SrcX(a0)
                        move.w  #0,bsa_SrcY(a0)
                        move.w  #400,bsa_SrcWidth(a0)
```

etc. etc. Till all the arguments are filled.

Then we call it

```
                        movea.l gfxbase,a6
                        jsr     _LVOBitMapScale(a6)
```

Then it delays for 1 second:

```
                        Delay(50)
```

which is

```
                        move.l #50,d1
                        move.l dosbase,a6
                        jsr     _LVODelay(a6)
```

Next it just repeats the scaling a few times using different scaling values. I have not done a direct copy of this code. I did about 10 different scalings in the assembler version to try it all out.

The BitMap scaling code is very flexible. You can scale things to any ratio within about 1:16000 (Scale it 16000 times its size). Examples 1:2 (double) 3:1 (one 3rd) 295:761 etc. etc. Its not made for doing demos or anything so don't think you can code a scaling demo with it.

After this it just frees the resources one at a time in the reverse order that we opened them. We do the same of course. 8-)

Well I hope this Volume of the Enconn Coders Manual has taught you something about reading C example code and I look forward to some system friendly demos from the demo coders scene in the near future! 8-)

********** 18. Startup&Exit ***********

Startup and Exit Problems with Amiga Coding

1 Your code won't run from an icon
2 Error codes on exit
3 Avoiding Forbid() and Permit()
4 Sprite problems

********** 18.1 Startup&Exit - Your code won't run from an icon ***********

Your code won't run from an icon

You stick an icon for your new demo (not everyone uses the CLI!) and
it either crashes or doesn't give back all the RAM it uses. Why?

Icon startup needs specific code to reply to the workbench message.
With the excellent Hisoft Devpac assember, all you need to do is add
the line

```
    include "misc/easystart.i"
```

and it magically works!

For those without Devpac, the relevent code is included in
this archive as iconstartup.i

********** 18.2 Startup&Exit - Error numbers ***********

Error numbers when run from CLI script files

You may get an error like this from your code when run from
a batch file:

```
Program failed return code 184641234.
```

The return code is the value in D0 at the end of your program, so
for a clean exit, always clear d0 immediately before your final
RTS.

Of course you can use the return code in your code to allow conditional
branching after your code in a script file. For example:

```
* Simple example - assemble to checkbutton
        opt     l-

        btst    #6,$bfe001      ; check left mouse button (hardware)
        bne.s   .notpressed
        moveq   #0,d0
        rts
.notpressed
        moveq   #5,d0
        rts
```

Assemble this, and you have a program that can tell if the mouse
button is pressed during bootup. Ideal for switching between startup
sequences, for example with this amigados script file.

```
checkbutton
if WARN
execute s:startup-nomousepressed
else
execute s:startup-mousepressed
endif
```

********** 18.3 Startup&Exit - Avoiding Forbit() and Permit() **********

Avoiding Forbid() and Permit()

I've tried it, this works, it's wonderful.

Instead of using Forbid() and Permit() to prevent the OS stealing time from your code, you could put your demo or game at a high task priority.

The following code at the beginning will do this:

```
        move.l  4.w,a6
        sub.l   a1,a1            ; Zero - Find current task
        jsr     _LVOFindTask(a6)

        move.l  d0,a1
        moveq   #127,d0                 ; task priority to very high...
        jsr     _LVOSetTaskPri(a6)
```

Now, only essential system activity will dare to steal time from your code. This means you can now carry on using dos.library to load files from hard drives, CD-ROM, etc, while your code is running.

Try using this instead of Forbid() and Permit(), and insert a new floppy disk while your code is running. Wow... The system recognises the disk change.... But remember to add your input handler!!!

Of course this is purely up to you. You may prefer to Forbid() when your code is running (it makes it easier to write).

********** 18.4 Startup&Exit - Sprite Initialization ***********

Sprite Initialisation

Some people doesn't initialize the sprites they don't want to
use correctly. (This reminds me of Soundtracker.)
A common error is unwanted sprites pointing at address $0.
If the longword at address $0 isn't zero you'll get some funny looking
sprites at unpredictable places.

The right way of getting rid of sprites is to point them to an address
you for sure know is #$00000000 (0.l), and with AGA you may need to
point to FOUR long words of 0 on a 64-bit boundary

```
           CNOP   0,8
pointhere: dc.l   0,0,0,0
```

The second problem is people turning off the sprite DMA at the wrong time.
Vertical stripes on the screen are not always beautiful. Wrong time means
that you turn off the DMA when it is "drawing" a sprite.
It is very easy to avoid this.
Just turn off the DMA when the raster is in the vertical blank area.

Currently V39 Kickstart has a bug where sprite resolution and width
are not always reset when you run your own code.
See Fixing Sprites in AGA

********** 19. Tracker Problems **********

Tracker Music Play Routines

No demo would be complete without a tracker module playing
in the background. It's a pity then that tracker code
seems to be amongst the worst written I've seen:

Protracker Replay code bug

I've just got the Protracker 2.3 update, and the replay code (both
the VBlank and CIA code) still has the same bug from 1.0!

At the front of the file is an equate

```
>DMAWait = 300 ; Set this as low as possible without losing low notes.
```

And then it goes on to use 300 as a hard coded value, never refering
to DMAWait!

Now, until I can get some free time to write a reliable scanline-wait
routine to replace their DBRA loops (does anyone want to write a better
Protracker player? Free fame & publicity :-), I suggest you change
the references to 300 in the code (except in the data tables!) to
DMAWait, and you make the DMAWait value *MUCH* higher.

I use 1024 on this Amiga 3000 without any apparent problem, but
perhaps it's safer to use a value around 2000. Amiga 4000/040
owners and those with 68040 cards tell me that between 1800 and
2000 are reasonable values...

There is a better Protracker replay routine in the source/ drawer.

********** 20. Video Standards **********

Coping with different Video Standards

As an European myself, I'm naturally biased agains the inferior NTSC system, but even though the US & Canada have a relatively minor Amiga community compared with Europe (Sorry, it's true :-) we should still help them out, even though they've never done a PAL Video Toaster for us (sob!).

You have two options.

Firstly, you could write your code only to use the first 200 display lines, and leave a black border at the bottom. This annoys PAL owners, who rightly expect things to have a full display. It took long enough for European games writers to work out that PAL displays were better.

You could write code that automatically checked which system it is running on and ran the correct code accordingly:

Two pieces of check code are needed. One handles the simple NTSC or PAL differences under Kickstart 1.2/1.3, but under Kickstart 2.0 or higher, everything is complicated by new monitor types...

1.3 Check for NTSC/PAL

```
        move.l        4.w,a6          ; execbase
        cmp.b        #50,VBlankFrequency(a6)
        beq.s        .pal

        jmp        I'm NTSC
.pal        jmp        I'm PAL
```

2.x/3.x Check for NTSC/PAL

```
        move.l  GfxBase,a6
        btst        #2,gb_DisplayFlags(a6)  ; Check for PAL
        bne.s   .pal

        jmp        I'm NTSC
.pal        jmp        I'm PAL
```

This test *may* work under 1.3, but the code in Kickstart 1.2/1.3 rom is totally broken, so it can guess wrong about NTSC/PAL quite often!

Check startup.asm for a way to combine the two tests together...

This is fine *EXCEPT* for one thing... It only tells you what video system the system was booted under. If you have a PAL machine and you run a 60hz interlaced workbench (for less

flicker) it's fine because the demo still runs in 50hz (as
long as your system runs from 50hz power).

However, NTSC owners can lose out, because if their display
is capable of PAL (by running a PAL fixer or running a PAL
display mode) this code completely ignores them and runs
NTSC anyway, however, if NTSC users select PAL from their boot menu
(2.x and 3.0 only) then it will work.

For demos and games you'd probably only want to run 50Hz anyway..

Now, if you want to force a machine into the other display system
you need some magic pokes: Here you go (beware other bits in
$dff1dc can do nasty things. One bit can reverese the polarity
on the video sync, not to healthy for some monitors I've heard...)

To turn a NTSC system into PAL (50Hz)

```
        move.w        #32,$dff1dc                ; Magically PAL
```

To turn a PAL system into NTSC (60Hz)

```
        move.w        #0,$dff1dc                ; Magically NTSC
```

Remember: Not all displays can handle both display systems!
Commdore 1084/1084S, Philips 8833/8852 and multisync monitors
will, but very few US TV's will handle PAL signals.

It might be polite for PAL demos to ask NTSC users if they
wish to switch to PAL (by the magic poke) or quit.

********** 21. Books **********

Good Books for Programmers

I've been asked to suggest some good books (why? Isn't howtocode good enough? ;-)

Hardware Reference Manual

Essential for demo and game coders.

Rom Kernal Manual: Libraries

Essential for *ALL* Amiga Programmers

Rom Kernal Manual: Devices

Essential if you plan to do any work with Device IO (input.device, timer.device, trackdisk.device, etc...)

Rom Kernal Manual: Includes & Autodocs

These are available on disk instead, which is a lot cheaper! Essential reference work,

All these books are available to developers on the CATS CD 2 as AmigaGuide files.. $50 from CATS US.

Amiga User Interface & Style Guide

Probably the most boring book I've ever read :-) Useful if you intend to write applications, but even then some of the rules have changed for V39 since this book was printed.

AmigaDOS manual 3rd Edition (Bantam)

Truly awful book, unfortunately the ONLY official dos.library reference. Why it can't be integrated into the RKM's I don't know... If you need to program dos.library and want info on AmigaDos file and hunk formats, this is the book.

Mapping the Amiga (Compute)

One of my favourite books. This is an easy-to-read reference to all system (1.3) functions and structures. Much easier to use than the Includes & Autodocs. I wish there was a V39 update to this!

Amiga System Programmers Guide (Abacus)

Quite handy, it covers a lot of the Hardware Reference manual, Rom Kernal Manuals and more in one book, but I'd suggest you buy the official books instead.

Advanced Amiga System Programmers Guide (Abacus)

Slightly more interesting than the first one, covers mainly

OS level programming, but again nothing really new.

Amiga Disk Drives Inside and Out (Abacus)

AVOID THIS BOOK! It has some of the worst code and coding practices
I have ever seen in it. Half of the code will only work under
Kickstart 1.2, the other half doesn't work at all!!!!

Amiga Realtime 3d Graphics (Sigma)

Wow! What a useful book! Explains how to do vector graphics
in 68000 on the Amiga, from basics to complex world models with
lightsourcing. I haven't tried any of the code from the book
yet, but it looks very good.

680x0 Programming by Example (Howard Sams & Company)

Excellent book on 68000 programming. Covers 68000/020/030
instructions, optimization. Aimed at the advanced 68000 user,
some really neat stuff in this book. The only 68000 book I've
bought, except the Motorola manual.

Motorola User's Manuals (Prentice Hall?)

Explains most things you'd ever wish to know about your
processor, and worth gold when trying to estimate
cycles used by your code and what commands there are
and everything. Thanks to ITT Multikomponent for giving
me the mc68030 User's Manual for free (why?).

The Discworld Series (Terry Pratchett)

Nothing to do with Amigas, but excellent books. If you need
a break from programming, or just inspiration for impossible
things to have in an impossible demo, read one of these!

Oook!